PIPELINED APACHE HTTP SERVER

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements of the Degree

Master of Science in Computer Science

by

Kevin Dinh Le

April 2009

COMMITTEE MEMBERSHIP

TITLE:	Pipelined Apache HTTP Server
AUTHOR:	Kevin Dinh Le
DATE SUBMITTED:	April 2009
COMMITTEE CHAIR:	Aaron Keen, Ph.D.
COMMITTEE MEMBER:	Michael Huangs, Ph.D.
COMMITTEE MEMBER:	Mei-Ling Liu, Ph.D

ABSTRACT

Pipelined Apache HTTP Server

Kevin Dinh Le

Web servers often become overloaded with irregular surges in web traffic. Several techniques have been explored to cope with these overloads such as distributing load throughout different servers. This thesis presents Pipelined Apache HTTP Server, a modified version of the Apache Software Foundation's HTTP Server[1] that utilizes a pipelined execution of Apache's request cycle. We discuss Apache's original architecture, the modifications necessary for implementation of pipelined execution, and analyze its run time. Ultimately, we hoped to increase throughput of Apache but fall short because of unbalanced request phases and pipelining overhead.

[1] This paper uses "Apache" as shorthand for "Apache HTTP Server."

ACKNOWLEDGMENTS

I would like to express my gratitude to all those who gave me the possibility to complete this thesis.

I am deeply indebted to my advisor, Professor Dr. Keen for guiding me through this thesis. I am obliged to Professor Dr. Haungs for his idea for this project. I am also thankful for Professor Dr. Liu for serving on my committee.

I would like to give my special thanks to my fiancée, Jen, whose love and patience enabled me to complete this work.

TABLE OF CONTENTS

LIST OF FIGURES

1 Introduction

Popular web sites often get overloaded with disproportionate increases in traffic. For example, web servers hosting special content, such as the NASA Mars Pathfinder landing, Olympic Games, or special commercial promotions, may get bogged down by a large number of clients trying to access this information at the same moment. Several methods have been explored to try to manage these surges, including systems that distribute client requests to multiple servers that host replicated information [Cardellini, Li].

The solution proposed by this thesis employs the concept of pipelining. A pipeline consists of a number of processing elements connected together in series. Each element in a pipeline can process information concurrently. A properly implemented pipelined solution may bring higher overall throughput for the system.

Current applications, including hardware architectures such as microprocessors and graphics processors, obtain increased throughput by utilizing a pipelined execution. In contrast, our approach leverages pipelining in software.

We chose to pipeline Apache HTTP Server [Apache] for many reasons. Firstly, we take advantage of its request cycle's phases. These distinct phases make it simple to arrange into pipeline stages. Plus, the phases in Apache's request cycle can be further divided to better fit pipelined execution if needed.

Secondly, in addition to community support from open source developers, the Apache Software Foundation documents their HTTP server's API very well; thus, this helps with modification of their request phases.

Furthermore, direct performance gains would definitely benefit web users in general because 68% of web servers currently run Apache [Apache].

Lastly, not only can Apache's HTTP Server benefit—any other HTTP server may take advantage of executing request phases in a parallel pipeline to achieve better throughput as well [Yao].

This paper is organized as follows. The next section presents the related work in dealing with web server overload. Section 3 goes over the relevant elements of Apache that need to be understood in order to implement a pipelined execution. Subsequently, Section 4 overviews the concept of pipelining. After that, the fifth section introduces our specific modifications to Apache. Then Section 6 displays an analysis of our results. Section 7 discusses future work to Pipelined Apache. And lastly, section 8 provides concluding remarks about our solution.

2 Related Work

Web servers often become overloaded with explosive increases in web traffic. This section describes several techniques that have been developed to improve web server performance.

2.1 Distributed Architectures

The overall performance and resource utilization of these servers can be improved by spreading document requests among a group of web servers. These systems distribute their data throughout server nodes by either replicating their content on one server's local disk or using a distributed file system.

Cardellini et al. [Cardellini] surveys several such distributed architectures. Most the methods examined provide transparency to users. In other words, the distributed system appears as a single server to clients. Cardellini et al. argue that non-transparent techniques have less utility because they do not allow the server system to control request distribution as well as requiring client/user interaction or software modification.

We briefly describe some of these approaches in the next subsections.

2.1.1 Client-based Approach

These techniques route requests from a client, such as a web browser, to a server that holds replicated data.

In one approach, client-side browsers route HTTP requests to replicated servers using client-generated addresses. An older browser, Netscape, used this technique by internally translating requests bound for www.netscape.com to another URI such as www1.netscape.com or www2.netscape.com. This method has limited applicability because it requires the client to be aware of the replicated server addresses, which may entail software updating if the addresses of the servers change.

A second client-based approach uses client-side proxies—intermediaries between clients and web server nodes that route client requests to known replicated servers. This mechanism has also limited applicability because it needs modification to already existing Internet components.

2.1.2 DNS-based Approach

As opposed to client-side approaches mentioned before, DNS-based approaches distribute load using a server-side cluster DNS (Domain Name System) server responsible for translating symbolic site names (i.e. URIs) to the IP addresses of replicated servers.

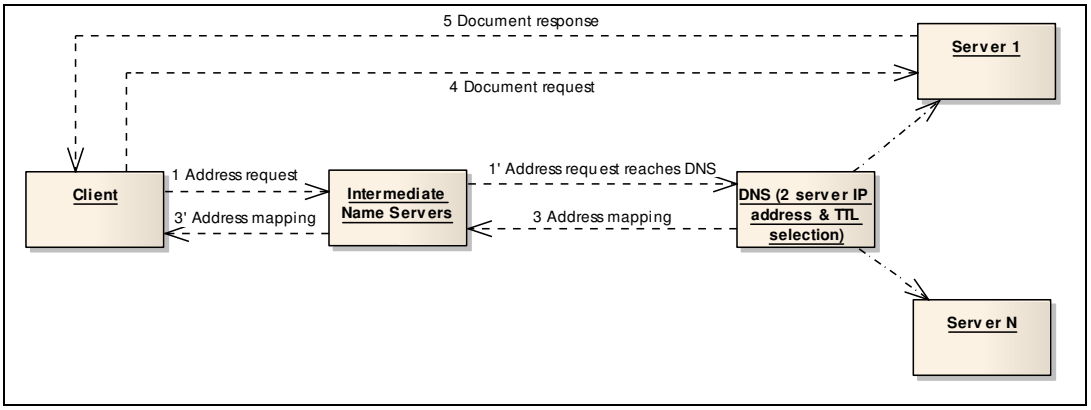

Figure 1: DNS-based Approach

This approach has limited control because intermediate server caching may cause these DNS servers to hold outdated IP addresses and thus, bypass the routing of the DNS. To counteract the mentioned caching, a time-to-live (TTL) counter can be attached from the DNS. When a client requests the resource from an intermediate caching server before the TTL for that resource expires, it simply returns the cached record. Otherwise, for a request for a resource with an expired TTL, the DNS finds an unloaded server node to return.

Cardellini et al. classify these methods based on the scheduling algorithm the DNS uses for server node selection. These algorithms can use either a constant TTL algorithm or an adaptive TTL algorithm. The next subsections describe each in more detail.

2.1.2.1 Constant TTL Algorithms

Constant TTL algorithms assign the same TTL value to all requests. The can either use no data, server state information, client state information, or both to schedule mappings to server nodes.

- **System-stateless Algorithms:** These algorithms do not use any system state to make the decision of which server node to forward a request. One such algorithm, the Round Robin DNS approach, simply rotates selection between each server node in the system. This method may lead to unbalanced server load if clients use address caching to bypass the DNS on subsequent requests.
- **Server-state-based Algorithms:** Knowing server state information such as capacity and availability can be useful in routing requests. For example, the lbmnamed algorithm selects the least loaded server to send requests to. It also avoids client side address caching by setting TTL values to zero so that all requests must go through the DNS. These algorithms can exclude unreachable servers because of fault or congestion as well.
- **Client-state-based Algorithms:** These algorithms can use two different types of client-side data: typical load from each connected domain and a client's geographic location. One such algorithm, the multi-tiered round-robin policy, uses a different server selection chain based on the average number of requests from a particular domain. Other algorithms use client location such as topological proximity, client-to-server link latency, and/or round-trip delays. Like server-state-based algorithms, the DNS sets the TTL to zero to get around client-side caching.

- **Server-and-client-state-based Algorithms:** The most effective DNS-approach algorithms combine both server and client state data, such as server availability information, client proximity, etc.

Cardellini et al. state that these constant TTL algorithms cannot sufficiently address the skew from client requests from the same domain and the problem of different server node capacities. The next subsection describes adaptive TTL algorithms, which tackle those concerns.

2.1.2.2 Adaptive TTL Algorithms

In contrast to constant TTL algorithms, adaptive TTL algorithms control the selection of server nodes through both the scheduling algorithm and dynamically modifying TTL values based on information from servers and/or clients.

To address client request skew, adaptive TTL algorithms DNS servers assign different TTL based on average request load from a domain. If clients from specific domains make large amounts of requests, the DNS will set smaller TTL values so that those clients must go through the DNS again to get a new address of a different server node. Thus, subsequent requests will not overload previously selected server nodes.

As for the issue of heterogeneity of server capacities, adaptive TTL algorithms can easily scale because they only require dynamically gathered information such as the request rate from each client's domain.

2.1.3 Dispatcher-based Approach

In a dispatcher-based architecture, a centralized dispatcher completely controls the scheduling and routing of client requests to server nodes within the

system. These systems identify servers using a private address. Dispatcher-based approaches differ by routing mechanisms.

One such mechanism, **packet rewriting**, alters the IP address of incoming requests to that of an unloaded server node's address. When the server returns a response to the client, it writes the return address as the address of the dispatcher.

Similarly, in **packet double-rewriting**, instead of directly returning a response to a client, the chosen server node returns it to the dispatcher. Likewise, the dispatcher must modify the IP address of the server's return response to that of the single address of the dispatcher before sending the response to the client.

Another dispatching solution uses **packet forwarding** to directly route requests to unloaded server nodes instead of rewriting IP addresses.

Moreover, a dispatcher can invoke **HTTP redirection** to specify an appropriate status code in an HTTP response's header to indicate which server the client can get a desired document. This method does not require altering IP addresses of packets.

Like previous approaches, dispatching solutions can take advantage of server state such as the load on each server. In choosing a server, dispatchers can also employ location-based dispatching to estimate a client's server proximity.

2.1.4 Server-based Approach

Server-based approaches borrow from the idea of dispatching but add another level. In these systems, a client makes a request to a DNS that assigns the request to a server node. Upon receiving a reply from the DNS, the client makes another request to the server specified by the address in the reply. When the server node receives the

client's second request, it can either process it or send it to another server node depending on the amount of load the current server has.

Like dispatcher-based systems, these techniques can use HTTP redirection or packet redirection (each described in the previous section).

2.1.5 Distributed Apache

Li and Moon have designed and implemented another distributed system using Apache servers to alleviate increased web traffic [Li]. Their Distributed Cooperative Apache (DC-Apache) web server system migrates and replicates documents among cooperating servers. It uses dynamic generation of hyperlinks to spread out requests and balance load. DC-Apache keeps a consistent state by utilizing a strict maintenance policy on each server node's replicated documents. In the end, Li and Moon show that their system provides better performance and scalability by distributing loads across multiple Apache servers.

2.2 Individual Server Node Optimization

In contrast to the above solutions, Pipelined Apache makes an attempt to mitigate load by improving the performance of an individual server node, as opposed to creating or modifying a distributed server system. Thus, system administrators in charge of a distributed web server system can make use of a pipelined server to supplement their current layout.

One such system, Yao et al.'s Pipeline [Yao], prototypes the pipeline architecture with their own HTTP server. Similarly to Apache, their implementation has a request pipeline consisting of the following phases.

- Accept
- Read request
- Analyze
- Read file
- Send file

Each stage of their pipeline follows a specific algorithm for each phase. This algorithm begins with a while loop. Then when a stage receives a "data block," it processes a portion of the request. Next, it hands the data block off to the next stage in the request cycle. Notice each stage uses a queue to buffer data blocks between stages. The pseudocode below outlines this behavior.

```
while (true) {
    queue_pop(this_stage_queue, &data_block);
    process(data_block);
    queue_push(next_stage_queue);
}
```

Yao et al.'s results seem promising because they indicate that their pipelined server outperforms a thread pooled one.

3 Apache HTTP Server Architecture

The previously mentioned related work dealt with web servers in general. In this section, we describe the Apache HTTP Server, the specific web server we modify to utilize a pipelined architecture. We outline the main components needed to be understood before applying a pipelined design.

3.1 Overview

Apache HTTP Server belongs to a category of programs called web servers. Clients use the HTTP protocol to send requests to Apache, which then uses the same protocol to send responses back. We outline the HTTP protocol in Section 3.2.1.

As shown in Figure 2: Basic web serverFigure 2, the fundamental functionality of a web server consists of the following steps:

1. Listening to open sockets for an HTTP request from a client (such as a web browser, spider, or other end-user tool)
2. Parsing the request
3. Generating an HTTP response based on the request
4. Returning the response to the client.

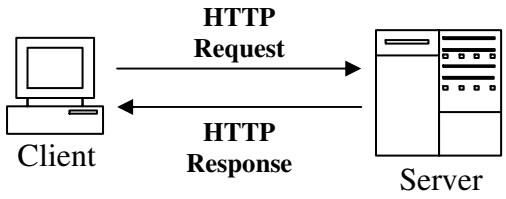

Figure 2: Basic web server

3.2 Request Phases

Once Apache accepts a client HTTP request, it processes the request using a cycle that prepares the HTTP response. Figure 3 shows the main phases of that cycle as described by the Apache Software Foundation.

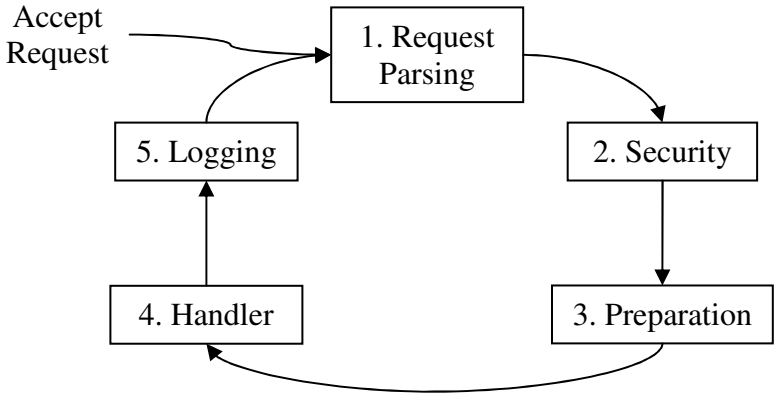

Figure 3: Apache Request Phases

3.2.1 Request Parsing Phase

The first part of Apache's request cycle, the request parsing phase, parses requests after Apache accepts them from a client. According to the HTTP protocol, an HTTP request consists of the following:

- A request line, which contains what information the request wants and where to locate it on the server as well as which HTTP version to use. For example, a request line to get a file named index.html using HTTP version 1.1 may consist of:
    - `GET /index.html HTTP/1.1`
- A request header, which may consist of, but not limited to, any of the following fields:
    - From
    - Accept
    - Accept-Encoding
    - Accept-Language

    - User-Agent
    - Referrer
    - Authorization
    - Charge-To
    - If-Modified-Since
    - Pragma
- An empty line
- An optional message body, usually for user-entered data or uploading files

Apache parses this information and appropriately determines what to do with it. In most cases, Apache translates the provided request line to the server's file system and then maps this location to a specific file. Sometimes a client may request a document using a certain URI (e.g. /index.html), but the web server hosting that document may have multiple pages with different languages (e.g. index.html.en, index.html.de, etc.) for that single URI. In this case, the server uses the Accept-Language field of the request header to figure out which file to send to the client. As seen, a request header can have different effects on how a request gets processed.

3.2.2 Security Phase

Security checking occurs after request parsing in Apache's request cycle. Apache runs three types of security checking: authentication ID checking, authorization access checking, and other access checking.

- **Authentication ID checking** confirms the client's identity. Usually, the client provides a username and password before or with a request. On the server, identification checking can use various forms of authentication such as

password files, anonymous authentication similar to FTP, Berkeley database (DB) files, or database management (DBM) files.

- **Authorization access checking** verifies if the user is authorized (i.e. has permission) to access content at the current location, file, and/or directory on the server. Authorization mostly uses the same or similar mechanisms as authentication.
- **Other access checking** checks whether access to a particular resource has been granted or denied based upon special conditions. In other words, access control restricts access based on something other than the user's identity. For example, an administrator may want to set restrictions based on where a client requests come from.

3.2.3 Preparation Phase

After checking security, Apache commences preparation processing of a requested object, which consists of 2 sub-phases: determining MIME type and performing "fixups."

3.2.3.1 Determining MIME type

The first sub-phase determines the MIME (Multipurpose Internet Mail Extension) type of requested information. Originally, the MIME standard extended the format of email to support additional features such as non-ASCII character sets. Since then, MIME has been used to describe other general content.

The HTTP protocol uses MIME types to differentiate between different forms of information including text in foreign languages (that use different character

encodings), images, sound, or video. Depending on the requested document's MIME type, Apache may use different modules to process the request.

3.2.3.2 Performing "fixups"

In the second step of the preparation phase, Apache performs "fixups." The Apache Software Foundation documents these fixups as opportunities for modules to perform module-specific fixing of header fields. They admit this approach does not represent a clean solution, but it is sometimes the "only option."

Modules register functions as hooks in order to get called during the fixups sub-phase. Some of these hook functions include adding extra authentication information, adding certain metadata, deciding whether to enable more translation for sub-requests and/or redirects, and much more.

3.2.4 Handler Phase

The (content) handler phase finally serves the request by preparing the HTTP response to be sent to the client.

In Apache, handler modules control the creation of content asked by HTTP requests. For example, if a request asks for dynamic content, such as a CGI- or Perl-generated response, Apache's handler phase will use that respective module to generate the response instead of the default file mapping handler.

In contrast, if a request does not require any special content generation, Apache uses its default content generator, which simply returns a file mapped (usually a text/HTML file) from the request's Uniform Resource Identifier (URI) to the web server's file system.

In handling a request, Apache can generate one or more sub-requests during processing. For example, when a client makes a main request for a language dependent webpage, e.g. index.html, Apache internally invokes a sub-request to return the webpage in the proper language, e.g. index.html.en for an English version of the webpage. The handler phase takes care of creating such sub-requests and starting them on the request cycle.

Whether a request asks for a simple HTML file or dynamically generated content, Apache recognizes what to return and the action the server should take in order to send an appropriate response back to the client for a particular request using information from the previous phases.

3.2.5 Logging Phase

After the handler phase has sent a reply back to the client's browser, a logging phase captures pertinent information about requests and places it into log files. In these files, Apache keeps information such as each request's client source, URI, and time to process. This data helps web administrators make reports, find problems, or even optimize their servers.

3.3 Process/Threading Structure

Apache uses the same aforementioned request phases across different platforms; however, an Apache developer (or web administrator) can specify which Multi-Processing Module (MPM) to use when building (or deploying) Apache. These MPMs modify how the web server controls the spawning and management of threads

and processes. This section looks at some of the popular MPMs used in distributions of Apache.

3.3.1 Windows MPM

By default in Windows, Apache's main process spawns one child process. This child process then spawns a pool of threads. **Error! Reference source not found.** illustrates this model.

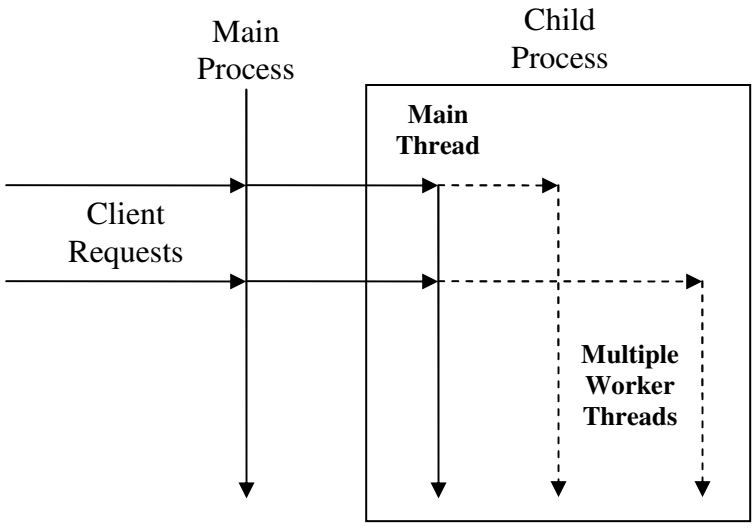

Figure 4: Windows MPM

Upon receiving a request, the main process forwards it to the child. Next, the child process's main thread looks for a free worker thread from the thread pool it created earlier and forwards the request to the worker thread. After the worker thread processes the request, the thread returns to the thread pool, awaiting further requests.

The Windows distribution uses threads instead of processes for enhancing performance because the Apache Foundation states that threads have less creation and context switching overhead than processes in Windows.

3.3.2 Prefork MPM

In UNIX on the other hand, since processes are more light-weight than in Windows, Apache trades a small performance hit for the added protection and fault-tolerance provided by processes. Thus, Apache uses the Prefork MPM by default on UNIX.

The Prefork MPM starts with a main process that "preforks" a set amount (specified by the user in a configuration file) of child processes. Once the main process forks the child process, it then listens for incoming connections from clients.

When a request comes in, the main process forwards the request to an idle child process. Once the child process finishes processing the request, it returns to the pool of processes and waits to process another request from the main process. Figure 5 Figure 5: Prefork MPMdisplays this progression.

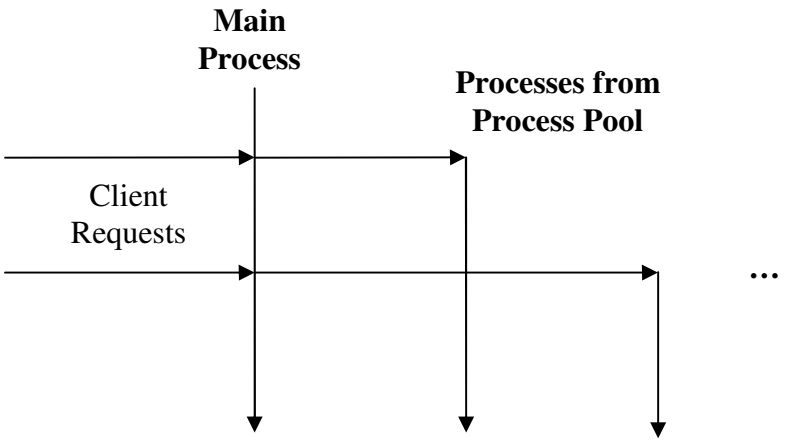

Figure 5: Prefork MPM

3.3.3 Worker MPM

In addition to the Windows and the Prefork MPM, the Apache developer can also choose to build Apache using the Worker MPM.

When using the Worker MPM, in addition to spawning multiple processes, each process also spawns a pool of spare threads. The server adjusts to different kinds of loads by dynamically increasing or decreasing the number of processes and also, the number of threads each process keeps spare. Figure 6 illustrates this design.

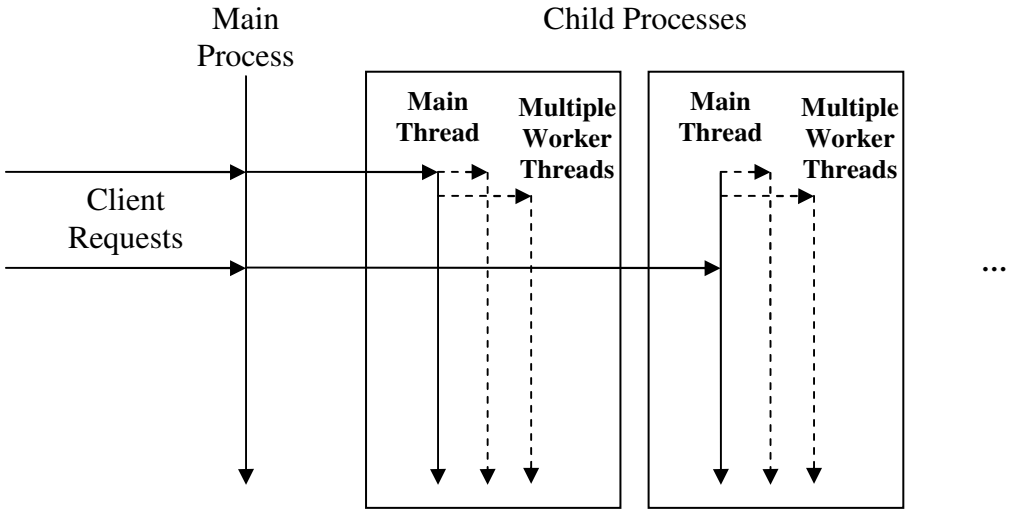

Figure 6: Worker MPM

3.4 Memory Management

No matter which MPM Apache employs for process control, every Apache build manages dynamic memory using memory pools. Resources initialized using these pools get freed when their respective pool becomes destroyed. For instance, memory allocated for a URI string created using a request's pool will get freed when the request completes. This simplifies development in that the programmer does not need to explicitly destroy all the resources constructed using a pool—just the one pool that groups those resources. More information on memory pools can be found in Appendix B (Section 10.2).

4 Pipelining

The solution proposed by this paper employs the concept of pipelining in which processing elements can concurrently deal with multiple tasks in a series of stages with the output of one stage in the pipeline getting passed on as input to another processing element in the pipeline. Some popular examples of pipelines include the following.

- **Instruction pipelines** in processors allow parallel execution of two or more instructions from a sequential stream. In the example of a RISC microprocessor's pipeline, logical circuits implement the various stages of an instruction (e.g. register fetching, address decoding, arithmetic/execution, cache/memory lookup, etc.).
- **Graphics processing pipelines**, found in most graphics cards, consist of multiple arithmetic units, or even complete CPUs, that simultaneously execute the various stages of common rendering operations (e.g. perspective projection, window clipping, color and light calculation, rendering, etc.).
- **Software pipelines** consist of multiple processes arranged so that the output stream of one process gets fed as the input stream of the next one. For example, in the UNIX environment, users can "pipe" output from one command line utility to another, such as the results of a directory listing (ls) piped to a word count program (wc) to get the number of files in a directory. Note that this sort of pipeline does not execute multiple tasks in parallel.

Pipelined systems require storage between individual stages to buffer incoming data. When one stage completes its task, it passes its product to another stage. If this

latter stage already has a current task, it must keep a buffer to store the previous stage's output. Note that the previous stage's output must wait in the latter stage's buffer until the latter stage completes its current task. It then follows that pipelining works best if the execution of an application can be divided into fairly equal time intervals so that an element in the pipeline does not have to wait for previous elements to complete to acquire data to process.

Tradeoffs exist when implementing a pipelined architecture. Generally, if its stages execute in parallel, a pipelined application sees a benefit of increased throughput—the number of tasks going through the system increases. However, the increased throughput does not come for free. Pipelining's extra communication and buffering between stages may cause increased latency. In other words, although more tasks may be finished in the end, it might take longer for a single task to be completed than if that task ran without pipelined execution. Therefore, pipelining may increase throughput but implementers need to be aware of the necessary overhead that causes increased response time.

Pipelining execution paths can also bring about performance increase by allowing multiple processing elements to process a single stage of the pipeline at the same time. For instance, in an instruction pipeline, the memory access stage might take longer to complete compared to other stages. Thus, a system designer might consider incorporating multiple memory stages that simultaneously execute to improve throughput. Conversely, this may introduce additional concerns such as implementation complexities and/or increasing the chances of making the pipeline less balanced.

5 Implementation

An efficient implementation of Pipelined Apache requires

- the balanced division of request phases,
- the addition of queues that buffer requests between stages of the pipeline, and
- the threading of each request phase.

The following subsections further elaborate on these requirements along with some unexpected problems.

5.1 Phase Balancing

A key aspect of pipeline design requires balancing pipeline stages so that processing elements further down the pipeline do not have to wait for previous processing elements to complete. Thus, in order to deal with this issue, we added timing code to an unmodified Worker build of Apache to record the time it takes to execute each phase of Apache's request cycle. We then conducted tests to figure out if Apache's request phases needed modification. Figure 7 shows our observations.

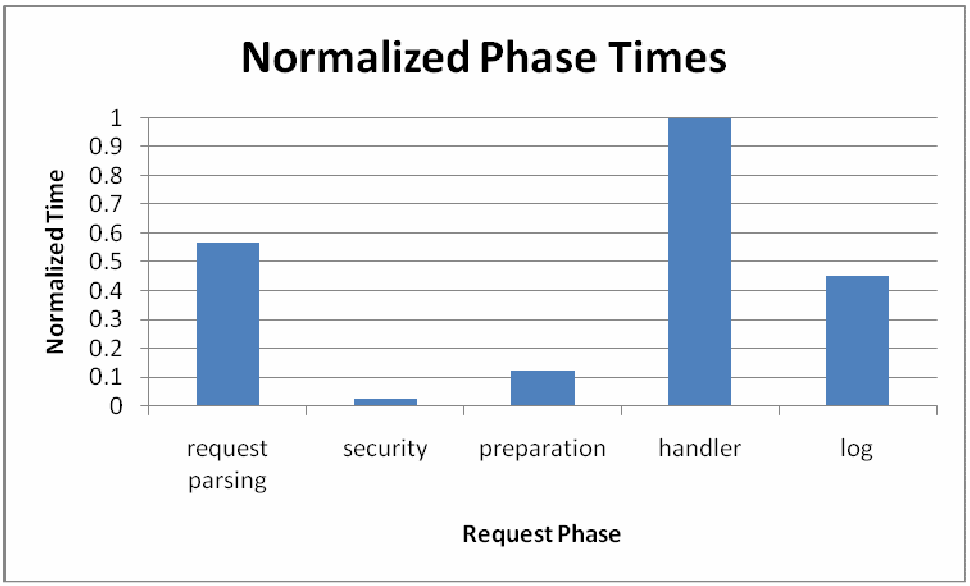

Figure 7: Phase timings

According to our timings, the request parsing and logging phase have comparable run times. On the other hand, the other phases (i.e. security, preparation, and handler) have very contrasting measurements. A more favorable, balanced solution would have one pipeline stage execute the security and preparation phases along with the log phase. However, this could not be accomplished because the handler phase divides the two segments. Thus, we chose to only combine the security and preparation phases into one stage.

As for the handler phase, we pondered separating it into smaller phases to closer match the processing times of the other phases. However, that task proved difficult because Apache uses different execution paths depending on the type of client request. In spite of that, we chose to attempt a simpler compromise, which would incorporate multiple handler stages in our pipeline.

5.2 Data Model

Once we decided upon how to integrate Apache's request phases into pipeline stages, we then looked at the data structures that Apache uses to manage requests. Apache processes a request by working primarily on a `request_rec` (short for request record) struct that holds information such as connection, memory, sub-request, URI, and other important data pertaining to that request. For each request in Apache, every function that does request processing modifies the request's `request_rec` struct directly by using a pointer to the struct. This greatly simplifies our implementation because we let Apache's included functions deal with request records and only invoke these functions in our pipeline's execution.

Even though we do not have to modify the way Apache parses and processes request data directly, we still need to make sure that data gets passed from one stage to the next in the pipeline. As mentioned before, pipelining requires the use of queues to buffer data between stages. We chose to utilize the included Apache Portable Runtime (APR) library since it provides blocking queues. Using these queues, a thread that calls `pop()` on a queue will block until data becomes available on that queue. In other words, the execution of threads does not waste valuable processor cycles when there no data exists in the queue to process. On the other hand, when data becomes accessible on a certain blocking queue, a thread waiting on that queue will unblock and start processing the available data.

5.3 Execution Model

After we figured out how to handle request data and how to partition the phases of Apache's request cycle, we needed to choose how these phases would run in a pipeline.

We wanted Pipelined Apache to execute like the Windows MPM (recall that the Windows MPM spawns a single process that in turn spawns a thread pool that handles incoming requests). However, we ran into some complications when trying to compile Apache to use the Windows MPM under Linux such as missing required Windows libraries. Despite that, we chose to modify the Worker MPM, which already runs similarly to the Windows MPM except that it dynamically spawns multiple processes, each with their own thread pool.

To accomplish the desired execution model, we added the ServerLimit directive in Apache's MPM configuration file and set it equal to 1 in an attempt to limit the Worker MPM to only spawn one process. However, even after setting this directive, we found that Apache would not allow the actual number of child processes to become any lower than two.

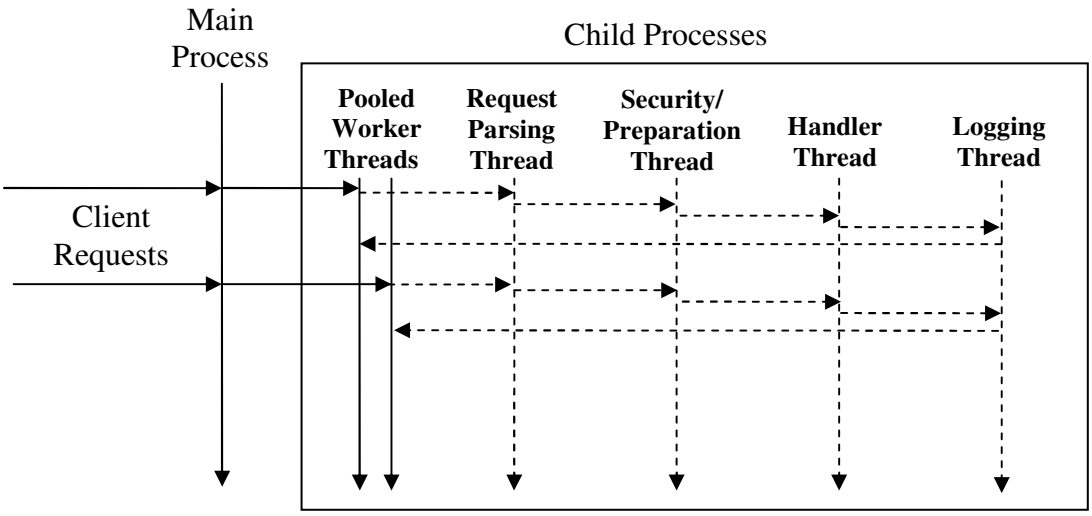

Figure 8: Execution of Pipelined Apache

Figure 8 diagrams Pipelined Apache running the modified Worker MPM. Unlike the Worker MPM, which spawns multiple processes, Pipelined Apache only spawns two child processes, almost similar to the Windows MPM. These child processes then spawn two types of threads, each explained below.

- First, 4 pipeline threads get spawned to execute each pipeline stage. After these pipeline threads spawn, they block until they receive a request (in the form of a `request_rec` struct pointer).
- Second, a thread pool gets spawned. These threads receive incoming requests from the main process.

With the goal of performance increase in mind, we had several reasons for choosing threads in our implementation. Firstly, threads share the same memory address space with each other, making programming much simpler without the usage of interprocess communications (IPC). The sharing of memory space also contributes to making the context switch penalties of threads less severe than that of processes

because of less cache misses. Additionally, threads have the ability to fulfill the requirement of running concurrently on separate processors as well.

5.3.1 Request Processing

Each stage of Pipelined Apache takes specific steps in order to process an incoming request. The following describes these steps along with showing the code call needed to accomplish those steps.

1. When a request arrives from a client, the main process forwards the request to a child process.

```
// in listener
apr_queue_push(worker_queue, socket);
```

2. Next, the child process hands the request off to an available thread in its thread pool.

```
// in worker
apr_queue_pop(worker_queue, &request_rec);
```

3. The thread sets up a mutex conditional to block until the request runs through the pipeline.

```
apr_mutex_cond_wait(mutex_cond, mutex);
```

4. The thread then places the request into the queue of the first stage of the pipeline (i.e. the request parsing phase).

```
apr_queue_push(pipeline->request_parsing_queue, request_rec);
```

5. The first stage was blocked, waiting for a request to arrive.

```
apr_queue_pop(pipeline->this_stage_queue, &request_rec);
```

6. Since a request became available on its queue, the thread can start processing the request. If it is not in the middle of processing a previous request, it will start processing the request immediately; otherwise, the request record waits inside the stage's queue until its turn for processing occurs.

```
do_stage_processing(request_rec);
```

7. Once the first stage of the pipeline finishes processing, it passes the request record to the next stage's queue.

```
apr_queue_push(next_stage_queue, request_rec);
```

8. This request record passing continues until it reaches the end of the pipeline where no more request processing can be done. At this point, the originally blocked thread can be signaled to continue.

```
apr_mutex_cond_signal(cond, mutex);
```

9. The original thread performs any necessary clean up and gets returned to the thread pool where it will block until awakened by the child process to execute another request.

5.3.2 Pipeline Stages

Most pipeline stages adhere to the following simple algorithm. When a pipeline thread spawns, it runs a while loop. It then calls `apr_queue_pop()` on the current stage's queue. This function blocks until a request becomes ready to be processed. Once dequeued, we call the actual function that does the processing. If that function returns an error, unlock the thread from the request thread pool that started this request and report the error. Else, the request gets passed on to the next stage's queue where it will follow the same (or similar) path to get processed. The pseudocode below outlines this process.

```
void pipeline_stage() {
    while (true) {
        // blocks if no elements in queue
        queue_pop(this_stage_queue, &request_rec)
        if (error = do_request_phase(request_rec)) {
            report error
        } else {
            queue_push(next_stage_queue, request_rec);
        }
    }
}
```

Most pipeline stages stick to the above plan. However, the preparation phase deviates because it spawns a sub-request that runs while the request that spawned it waits until it completes. This effectively blocks the preparation stage and causes deadlock. We detail this issue and our solution to this problem in the next section.

5.3.3 Preparation Phase Deadlock

As mentioned before, we ran into some trouble with the preparation phase. In the preparation of a request, Apache might find that that request requires a sub-request. In this case, Apache generates another request record for that sub-request. In our implementation, the sub-request gets placed onto the queue of the first processing stage (i.e. the Request Parsing phase) and continues down the pipeline. A problem arises, however, when the sub-request reaches the preparation stage where the main request has blocked the preparation processing. More succinctly, a deadlock occurs because the main request cannot complete until the sub-request finishes, and likewise, the sub-request cannot finish because the main request holds the security/preparation stage up while waiting for the sub-request. Figure 9 outlines this deadlock.

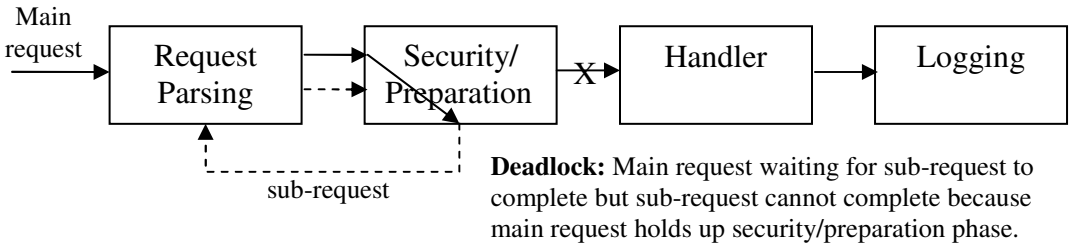

Figure 9: Preparation Phase Deadlock

Our first solution to this problem has the security/preparation stage spawn threads to process main requests. More accurately, when this stage pops a request off its queue, it determines what action to take based on the kind of request it encounters.

- **Main request**: Pass this request to a spawned thread to process. When the thread spawns a sub-request, it calls `ap_process_request_internal()`, which blocks the thread using a mutex conditional until the sub-request finishes.

- **Sub-request**: Have the preparation stage's thread process this sub-request. Once it completes, unlock its main request's mutex conditional.

The above approach fixed the deadlock but it hindered performance because of the spawning and destruction of threads. To counter the speed decrease, we modified the security/preparation stage to spawn a fixed number of threads at initiation instead of when requests come in. Figure 10 shows this process.

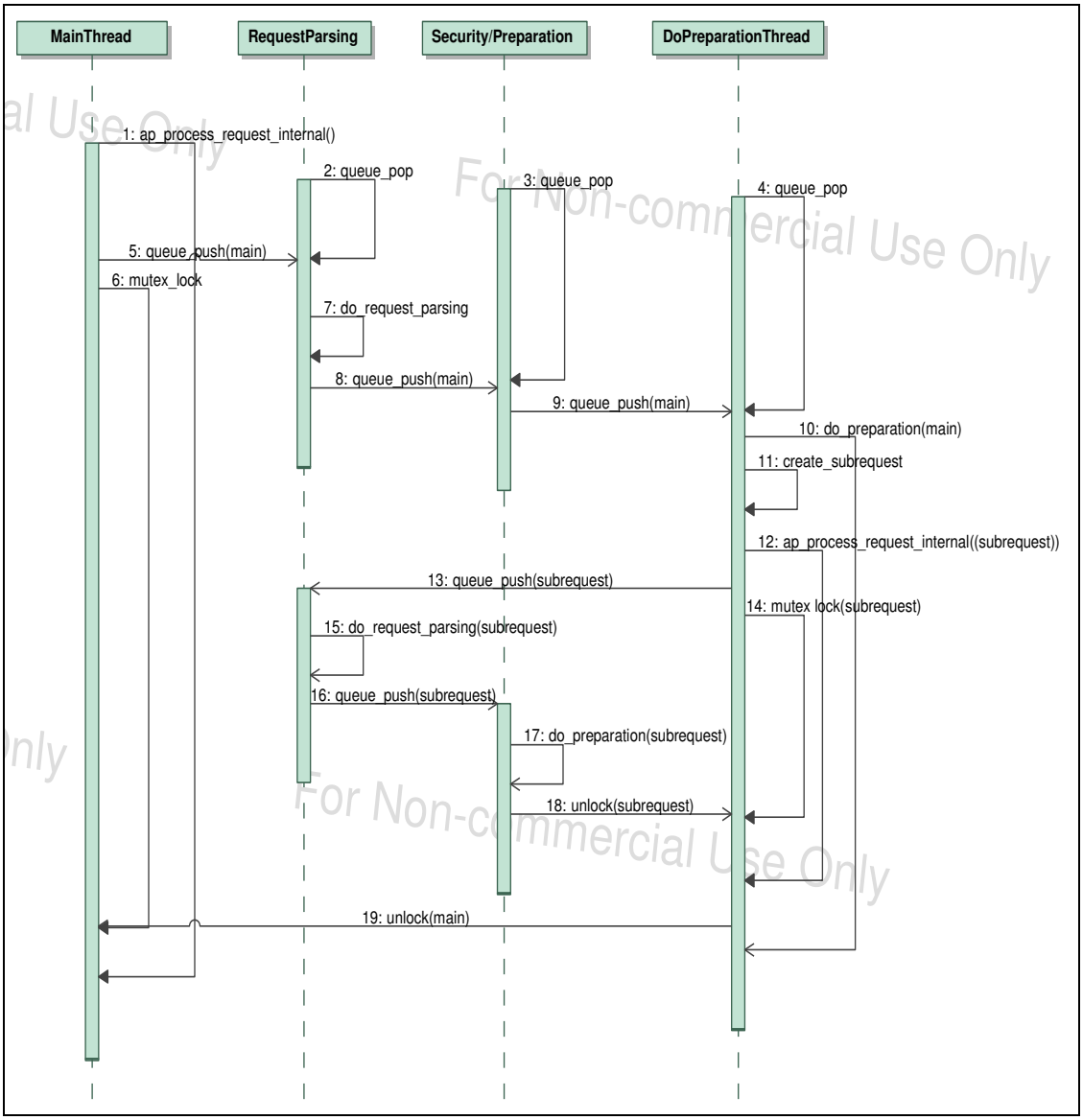

Figure 10: Preparation Processing Sequence Diagram

The code block below shows the necessary modification of the simple pipeline stage.

```
void general_request_stage() {

    while (true) {

        queue_pop(this_stage_queue, &request_wrapper); // blocks until
receives request

        do_request_phase(request_wrapper->request_rec);

        queue_push(next_stage_queue, main_request);
    }
}
```

Notice that the above pseudocode requires each request to keep a mutex conditional. In order to store a mutex conditional for each request, we create a new `request_wrapper` struct that wraps a request by keeping a pointer to it. With this technique, we can then append additional data to request records without altering Apache's original `request_rec` struct. A wrapper gets created when an incoming `request_rec` enters the pipeline. Henceforth, every pipeline stage accesses a request record by using the pointer contained inside that wrapper. We modified Apache's code as follows.

Original code:

```
// after created request_rec

queue_push(first_stage_queue, request_rec);
```

Modified code:

```
// after created request_rec

request_wrapper_t *request_wrapper;

request_wrapper_create(&request_wrapper, request_rec);

queue_push(first_stage_queue, request_wrapper);
```

Finally, the following pseudocode displays the needed steps to avoid deadlock for the preparation stage. (Unrelated to the deadlock but to avoid confusion, notice that the following function combines the preparation stage and the security phase.)

```
security_preparation_stage() {
    create thread pool to process main requests
    while (true) {
        queue_pop(this_stage_queue, &request_wrapper); // blocks until
receives request

        if (request is a sub-request or redirect)
            do_security_preparation for sub-request/redirect
            unblock main request's conditional in
                ap_process_request_internal()
        }
        else { //request is a main request
            queue_push(main_request__queue, request_wrapper)
        }
}
main_request_thread(main_request) {
    while (true) {
        // blocks until receives request
        queue_pop(main_request_queue, &request_wrapper);

        // blocks until complete. May spawn a sub-request, which will
        // block when it gets put into pipeline to be processed at
        // ap_process_request_internal()
        process(request_wrapper->request_rec)
    }
}
```

5.4 Pipeline Management

In order to control the life cycle of the pipeline in Pipelined Apache, we introduced a struct, `pipeline_t`. This struct holds pointers to key information, such as server configuration, memory pools, threads, and queues.

5.4.1 Pipeline Creation

During Apache's and subsequently, the MPM's startup, a single pipeline instance gets created and initialized by calling `pipeline_create()`. This function first constructs the necessary queues for buffering `request_rec`'s between each stage. In turn, it spawns threads for each stage in the pipeline.

```
// for each stage
apr_queue_create(this_stage_queue, capacity, pool);
apr_thread_create(this_stage_thread, thread_attributes, thread_arguments,
pool);
```

5.4.2 Pipeline Destruction

When Apache shutdowns, it performs various tasks such as outputting to log files. This happens when Apache receives an exit signal, which forces it into graceful shutdown mode. For Pipelined Apache's implementation, a complication arises because in order to gracefully exit, the current requests in the pipeline need to be completed. To accomplish this, we make the following changes.

- A variable keeps track of the current number of requests in the pipeline.
- We modified Apache so that when it receives the shutdown signal, a dummy request gets created and sent to each stage in the pipeline. When the stages receive this dummy request, they set a shutdown flag.

A pipeline stage can only exit when it has its shutdown flag set in addition to having no requests exist in the pipeline using the number of request variable. The following sequence diagram in Figure 11 shows this algorithm.

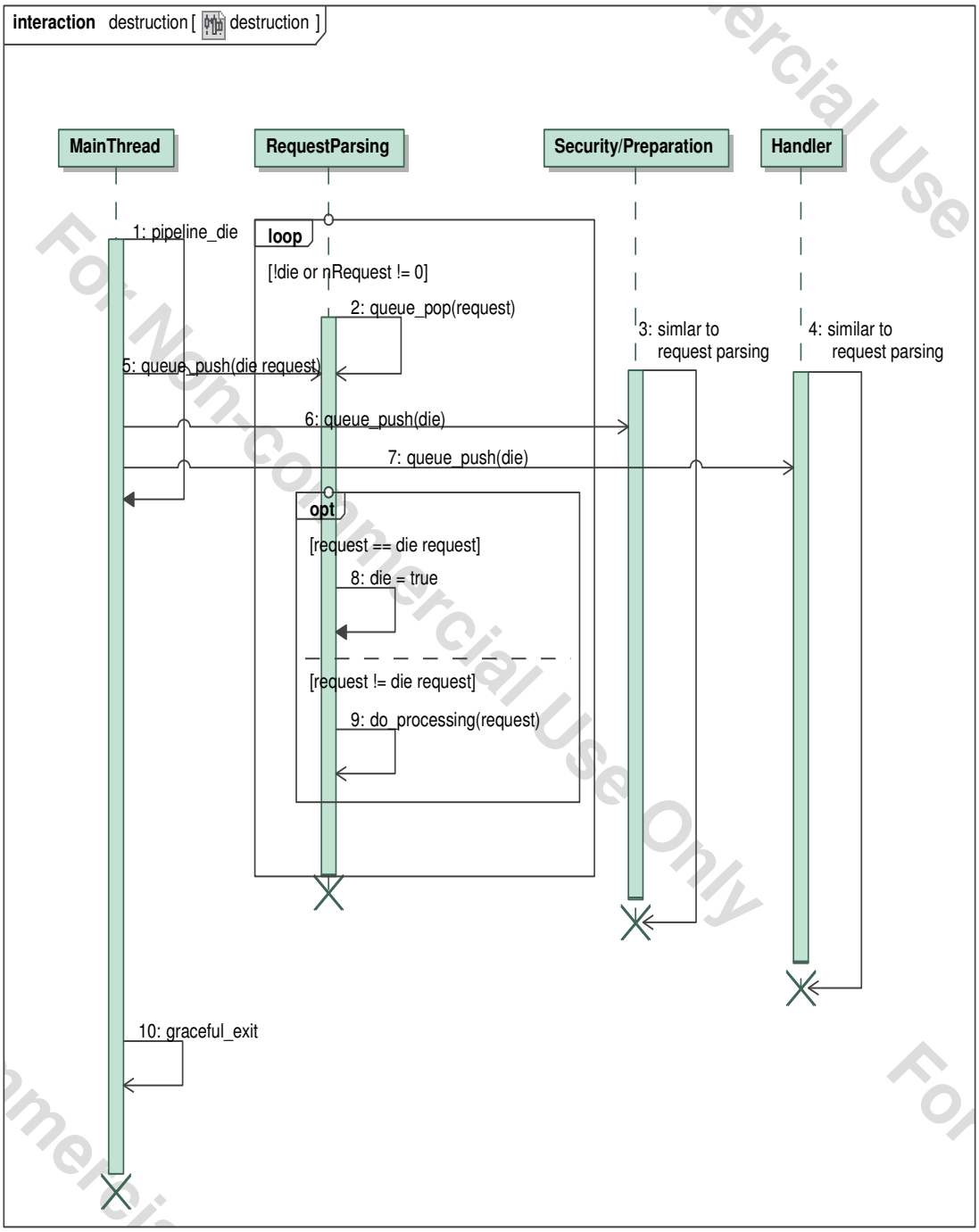

Figure 11: Pipeline Destruction

6 Results

After we implemented Pipelined Apache, we ran benchmarks on it to test its performance. We gathered these timing results and made comparisons to versions of Apache built with the unmodified Worker MPM and the unmodified Prefork MPM, both ran using default settings.

Our tests consisted of using the ApacheBench program, which comes included with Apache, to send concurrent requests for Apache's default test page. To gauge scalability, we increased the number of concurrent requests for each run by 16. Furthermore, since the results varied a bit from time to time, we executed ApacheBench seven times for each level of concurrency and then took their average.

For these tests, a computer with an Intel Core 2 Quad processor running 4 cores at 3.0 GHz with 4 GB RAM using a 10,000 rpm SATA hard drive ran the Worker, Prefork, and Pipelined variants of Apache 2.2.4 on Fedora Core 8.

6.1 Pipelined Apache Configuration

Apache uses several directives to configure its MPMs. In the case of the Worker MPM, the `ThreadsPerChild` directive controls the maximum number of spawned worker threads present in a child process. Since Pipelined Apache modifies the Worker MPM, it also uses this directive in the same way.

When setting the `ThreadPerChild` to a higher value than 64, Apache warned us that such a high value exceeded the `ThreadLimit` directive and that `ThreadsPerChild` would automatically be set to 64. In order to avoid this, we tried increased `ThreadLimit` to 150 and found that it did not affect performance positively or

negatively. On a side note, during our tests, the results fell significantly after about 150 concurrent requests as we suspect Apache may be nearing the `ThreadLimit` value mentioned before.

We experimented with different values of the `ThreadsPerChild` directive to try to optimize the performance of Pipelined Apache. Figure 12 charts the throughput of Pipelined Apache runs with various values of `ThreadsPerChild`. It shows that using a value less than 5 for `ThreadsPerChild` produces suboptimal performance. Secondly, any value between 5 and 85 results in similar timings. As the values of `ThreadsPerChild` increase, Pipelined Apache could handle more concurrent requests but at the cost of a much lower throughput. Thus, we chose to run the rest of our benchmarks with the default 25 `ThreadsPerChild` to get the best of both with high throughput and handling of a decent amount of requests.

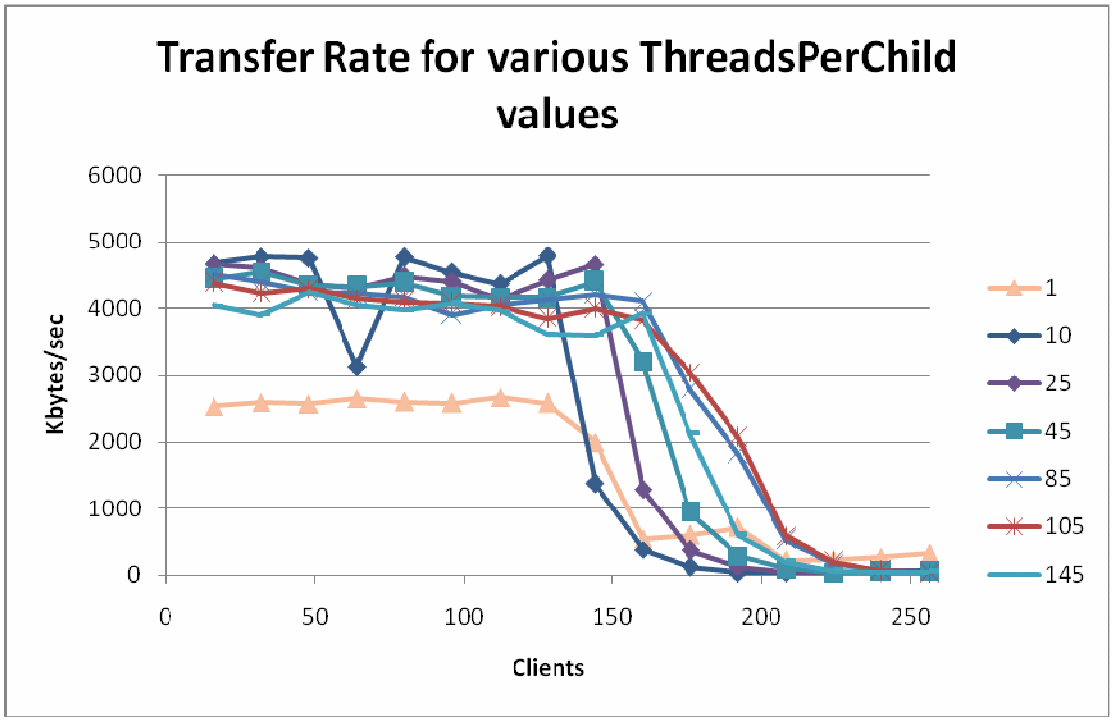

Figure 12: Transfer Rate of different ThreadsPerChild values

6.2 Pipeline Overhead

After finding the most favorable `ThreadsPerChild` value, we wanted to test if the added communications, buffering, and context switching overhead of pipelining increased the latency of a single request. To do that, we looked at another metric, request rate—the time it takes for a single request to complete. If the average request rate increased, we can attribute it to the pipeline overhead. Figure 13 shows the the average request rate for each Apache version.

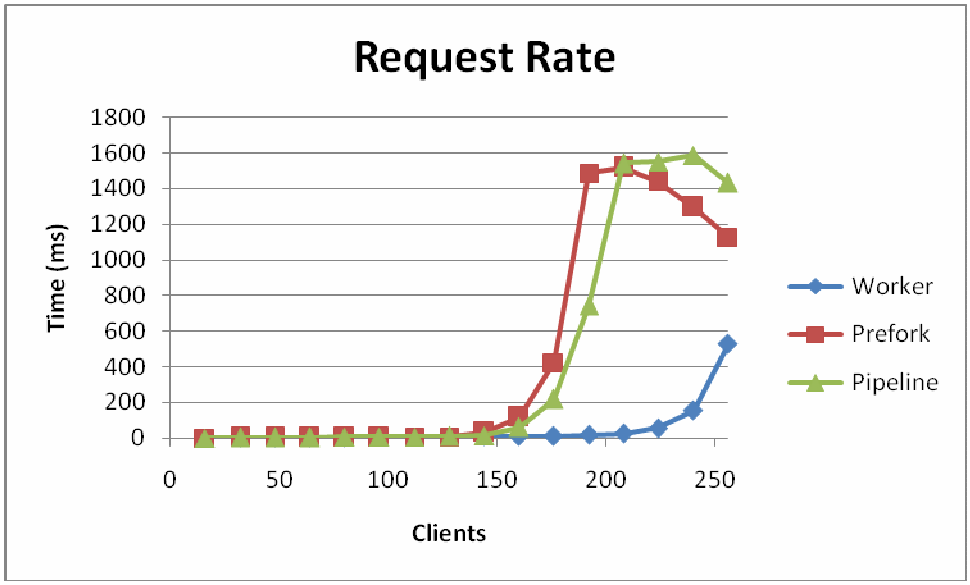

Figure 13: Request rate, mean

Figure 14 displays the average time to complete a request across all concurrent requests.

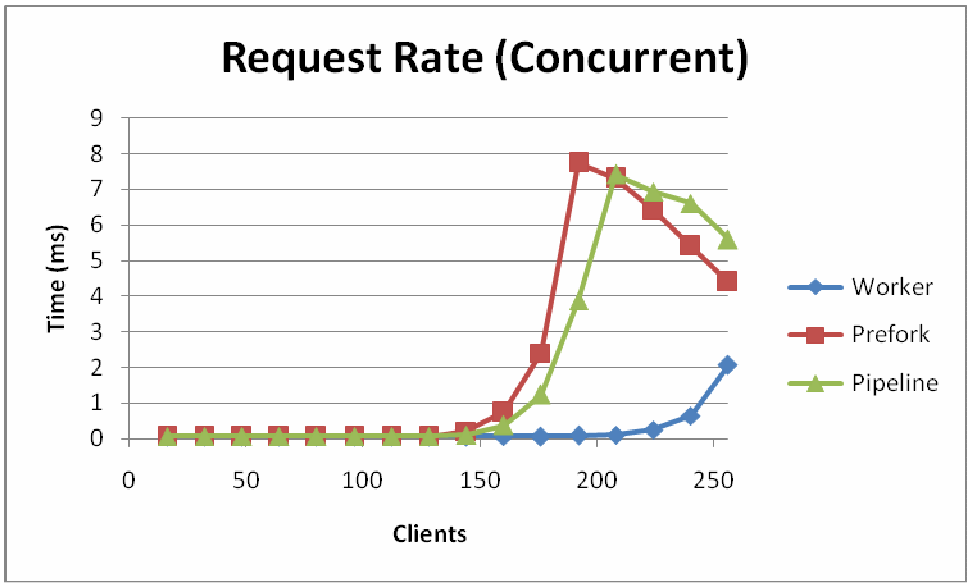

Figure 14: Request rate, mean across all concurrent requests

As expected, Pipelined Apache should have a higher request rate than its baseline, Apache built on the Worker MPM. Both graphs confirm our hypothesis of the increased overhead.

At a low number of concurrent clients, Pipelined Apache performs reasonably well, compared to Worker and Prefork, but the amount of time required to complete a request starts to climb significantly after 200 simultaneous clients.

6.3 Throughput

Overall, we intended to see if Pipelined Apache achieved improved throughput. To do that, we looked at how much data it could handle compared to the unmodified variants of Apache. Figure 15 shows the results.

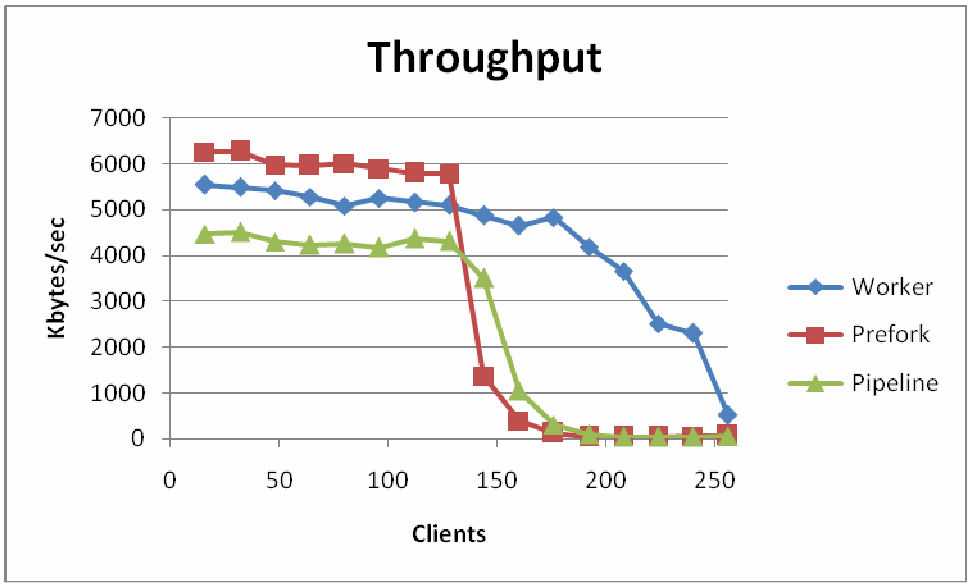

Figure 15: Transfer Rate

This chart reveals that Pipelined Apache unfortunately does not perform up to par with the Worker and Prefork MPMs.

6.4 Multiple Handler Stages

As our modifications did not gain performance, we drew up ideas to further increase performance. Recall from Section 5.1 that the handler phase ran in nearly double the times of the other phases in the request cycle combined. We hypothesized that adding more handler stages to our pipeline would make Pipelined Apache run faster. Figure 16 and Figure 17 show the results of adding multiple handler stages.

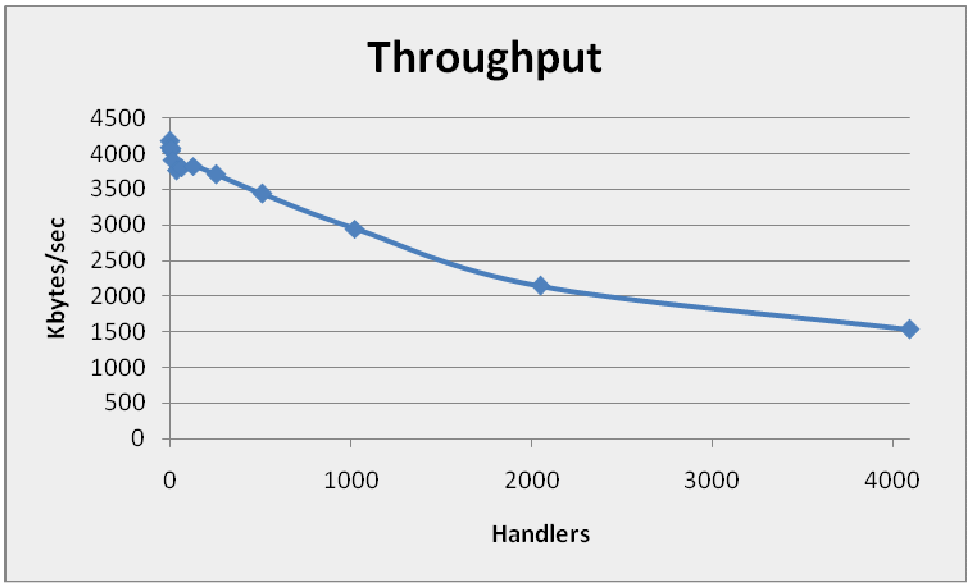

Figure 16: Throughput for various handlers quantities

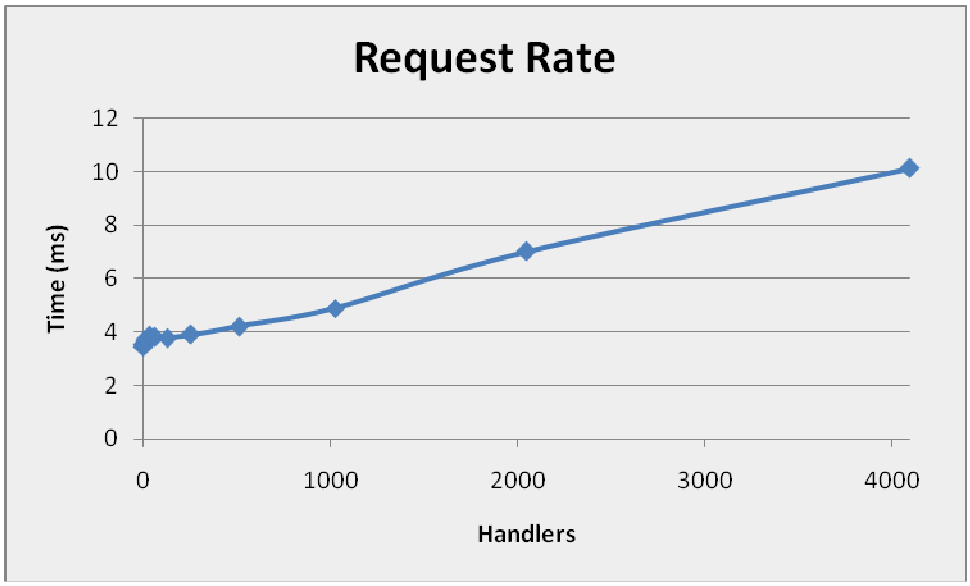

Figure 17: Request Rate for various handlers quantities

6.5 Results Summary

The above charts reveal that Pipelined Apache unfortunately does not perform up to par with the Worker and Prefork MPMs.

One plausible explanation for this drop may be that the stages in Pipelined Apache are simply not balanced enough. To make matters worse, this may be a problem that will be hard to overcome because of the variety of paths Apache takes when serving different types of requests.

Also, multiple handler stages actually decreased Pipeline Apache's throughput and increased the time taken for each request to complete. We conjecture that even before adding the multiple handlers, the large quantity of threads—request threads, pipeline stage threads, security/preparation threads, and now, handler threads—causes increased context switching between threads. (In a multitasking system, multiple threads and processes run in a time sliced manner to appear like all of them are being ran simultaneously to be more responsive. When the number of threads and processes exceeds the number of processors on a system, context switching occurs. The system must spend processor time to save out the current registers and file descriptors and load in new ones.) Thus, this context switching, which occurs because of the limited number of processors in our test system, may lead to Pipelined Apache's reduced performance. We suspect that a system with more processors would mitigate this penalty.

7 Future Work

As the above benchmarks show, Pipelined Apache's performance could use improvement. One task to achieve better execution times may be to further balance its request stages. However as mentioned before, this undertaking appears difficult because of the many execution paths Apache may take depending of the type of request given from a client. Also, if this deeper pipeline requires additional division

of the request phases, Pipelined Apache may need to run on a computer with even more processors to attain better performance to avoid thread context switching.

Another experiment to further examine the performance of Pipelined Apache can be realized by setting up a cluster of computers, each dedicated to running one specific stage of the pipeline. In doing this, Pipelined Apache must be modified so that each computer can become analogous to a pipeline thread in a single Pipelined Apache process. Nevertheless, we must be careful in implementing this distributed architecture such that the additional overhead of network communications does not slow down the overall system. Furthermore, this architecture's implementation and deployment may also prove to be difficult because of synchronization issues of served files, request records, etc.

8 Conclusion

Many efforts at optimizing web server systems have been made. These include distributed web server systems and individual server node optimization. This paper presents Pipelined Apache, an attempt at optimizing the Apache Foundation's HTTP Server by pipelining the phases of Apache's request cycle and having threads run these stages in parallel. A balanced pipelining of the execution of this request cycle can bring about increased throughput but at the cost of increased latency per request.

We have outlined Apache HTTP Server's framework, which accepts client HTTP requests and processes those requests through a cycle that prepares HTTP responses. This cycle includes header parsing, security checking, preparation, and handling of a request. We then explained the challenges of the implementation of Pipelined Apache including stage balancing, data forwarding, and thread execution.

Our implementation came short in performance compared to unmodified versions of Apache. We infer that this stems from two major shortcomings. First, Apache's request pipeline could not be effectively balanced. Attempting to further balance these stages proves difficult because of the complexity involved with different types of request handling in Apache. Second, the sheer amount of threads in its implementation caused Pipelined Apache to exhibit excessive context switch overhead. The trend of more processors in computer systems may make it possible for each of Pipelined Apache's large number of threads to be able to run on their own processor—eliminating this overhead. In the end, despite our limitations, we obtained a proof of concept by implementing an experimental architecture for an existing, popular web server.

9 References

[Apache] The Apache Software Foundation. http://www.apache.org/

[Cardellini] V. Cardellini, M. Colajanni, and P. Yu. "Dynamic load balancing on web-server systems", *In IEEE Internet Computing*, pages 28-39, May-June 1999.

[Li] Quanzhong Li and Bongki Moon. Distributed Cooperative Apache Web Server. *In Proceedings International WWW Conference*(10), Hong-Kong 2001.

[Yao] Nian-Min Yao, Ming-Yang Zheng, and Jiu-Bin Ju. Pipeline: a new architecture of high performance servers. *ACM SIGOPS Operating Systems Review*, v. 36 n. 4, pages 55-64, Oct. 2002.

10 Appendix A: APR Library

Apache provides the Apache Portable Runtime (APR) library for portable development between different platforms. This section describes relevant APR library functions used in the implementation of Pipelined Apache.

10.1 apr_status_t

A majority of Apache's APR library's functions return a status variable of type `apr_status_t`. This type actually typedefs an unsigned int. The following lists common return values and their descriptions.

10.1.1 APR Error Values

APR_ENOSTAT	APR was unable to perform a stat on the file
APR_ENOPOOL	APR was not provided a pool with which to allocate memory
APR_EBADDATE	APR was given an invalid date
APR_EINVALSOCK	APR was given an invalid socket
APR_ENOPROC	APR was not given a process structure
APR_ENOTIME	APR was not given a time structure
APR_ENODIR	APR was not given a directory structure
APR_ENOLOCK	APR was not given a lock structure
APR_ENOPOLL	APR was not given a poll structure
APR_ENOSOCKET	APR was not given a socket
APR_ENOTHREAD	APR was not given a thread structure
APR_ENOTHDKEY	APR was not given a thread key structure
APR_ENOSHMAVAIL	There is no more shared memory available
APR_EDSOOPEN	APR was unable to open the dso object. For more information

	call apr_dso_error().
APR_EGENERAL	General failure (specific information not available)
APR_EBADIP	The specified IP address is invalid
APR_EBADMASK	The specified netmask is invalid
APR_ESYMNOTFOUND	Could not find the requested symbol

10.1.2 APR Status Values

APR_INCHILD	Program is currently executing in the child
APR_INPARENT	Program is currently executing in the parent
APR_DETACH	The thread is detached
APR_NOTDETACH	The thread is not detached
APR_CHILD_DONE	The child has finished executing
APR_CHILD_NOTDONE	The child has not finished executing
APR_TIMEUP	The operation did not finish before the timeout
APR_INCOMPLETE	The operation was incomplete although some processing was performed and the results are partially valid
APR_BADCH	Getopt found an option not in the option string
APR_BADARG	Getopt found an option that is missing an argument and an argument was specified in the option string
APR_EOF	APR has encountered the end of the file
APR_NOTFOUND	APR was unable to find the socket in the poll structure
APR_ANONYMOUS	APR is using anonymous shared memory
APR_FILEBASED	APR is using a file name as the key to the shared memory
APR_KEYBASED	APR is using a shared key as the key to the shared memory
APR_EINIT	Ininitalizer value. If no option has been found, but the status variable requires a value, this should be used
APR_ENOTIMPL	The APR function has not been implemented on this platform, either because nobody has gotten to it yet, or the function is impossible on this platform.
APR_EMISMATCH	Two passwords do not match.
APR_EABSOLUTE	The given path was absolute.

```
APR_ERELATIVE      The given path was relative.
APR_EINCOMPLETE    The given path was neither relative nor absolute.
APR_EABOVEROOT     The given path was above the root path.
APR_EBUSY          The given lock was busy.
APR_EPROC_UNKNOWN  The given process wasn't recognized by APR
```

10.2 apr_pool_t

Most of Apache's APR library depends on memory pools. Memory pools manage a set of memory chunks. For instance, if a developer allocates a large number of memory chunks, he/she would need to call `free()` on that number or from memory leak bugs. Memory pools tackle this issue. As a result, memory pools force session-oriented programming.

```
/* declarations */
APR_DECLARE(apr_status_t) apr_pool_create(apr_pool_t **newpool,
                                          apr_pool_t *parent);
APR_DECLARE(void *) apr_palloc(apr_pool_t *p, apr_size_t size);
APR_DECLARE(void) apr_pool_destroy(apr_pool_t *p);
apr_pool_t *mp;


/* create a memory pool. */
apr_pool_create(&mp, NULL);


/* allocate memory chunks from the memory pool */
char *buf1, *buf2;
buf1 = apr_palloc(mp, MEM_ALLOC_SIZE);
buf2 = apr_palloc(mp, MEM_ALLOC_SIZE);


/* buf1 and buf2 get deallocated
apr_pool_destroy(mp);
```

Resources allocated using a single memory pool each have the same lifetimes. For instance, after allocating a memory pool, a developer can allocate multiple memory chunks from that pool. A single call to the memory pool destroy function deallocates all the memory allocated using that memory pool. As a result, a developer can manipulate the lifetime of an object using a memory pool.

10.3 apr_queue_t

Apache's portable runtime library provides thread safe blocking queues. A thread will block on a call to `apr_queue_pop()` if the queue has no elements. The thread unblock once an element because available on the queue. If the developer does not wish the thread to block, he/she can call call `apr_queue_trypop()`, which returns `APR_SUCCESS`.

```
/* declarations */
APR_DECLARE(apr_status_t) apr_queue_create(
        apr_queue_t** queue, unsigned int capacity, apr_pool_t* a);
APR_DECLARE(apr_status_t) apr_queue_push(apr_queue_t* queue, void* object);
APR_DECLARE(apr_status_t) apr_queue_trypop(apr_queue_t *queue, void **data);
APR_DECLARE(apr_status_t) apr_queue_pop(apr_queue_t *queue, void **data);
```

10.4 apr_thread_t

Apache's portable threading library works on several platforms by using an operating system's thread API. For example, under UNIX, `apr_thread_create()` calls `pthread_create()`.

For portability, `APR_THREAD_FUNC` macro is required.

```
APR_DECLARE(apr_status_t)
apr_threadattr_create(apr_threadattr_t **new_attr, apr_pool_t *cont);


APR_DECLARE(apr_status_t) apr_thread_create(apr_thread_t **new_thread,
                                            apr_threadattr_t *attr,
                                            apr_thread_start_t func,
                                            void *data,
                                            apr_pool_t *cont);


void* APR_THREAD_FUNC doit(apr_thread_t *thd, void *data);
```

10.5 apr_thread_mutex_t

Mutex's (or mutual exclusions) allow sole access to portions of code by only one thread at a time.

Like APR's threads, its mutex's use the platform's mutex's as well. The following functions make use of pthread_mutex's in UNIX.

```
/* declarations */
APR_DECLARE(apr_status_t) apr_thread_mutex_lock(apr_thread_mutex_t *mutex);
APR_DECLARE(apr_status_t) apr_thread_mutex_trylock(apr_thread_mutex_t
*mutex);
APR_DECLARE(apr_status_t) apr_thread_mutex_unlock(apr_thread_mutex_t
*mutex);


/* pseudo code: mutex lock's typical code */
/* case one */
apr_thread_mutex_lock(mutex); /* sleep until current thread acquires the
mutex lock */
```

```
do something on shared objects  /* during this, other threads can't acquire
the mutex lock. */
apr_thread_mutex_unlock(mutex); /* unlock ASAP after lock */


/* case two */
apr_status_t rv = apr_thread_mutex_trylock(mutex);
if (APR_STATUS_IS_EBUSY(rv)) {
    go through /* no need to unlock, because didn't acquire the lock */
} else {
    do something on shared objects
    apr_thread_mutex_unlock(mutex); /* should unlock */
}
```

10.6 apr_thread_cond_t

Thread conditionals allow blocking and signaling to wake up a thread.

When creating a thread conditional, a developer must pass a mutex that will lock upon blocking and signaling the conditional.

Similarly to APR's threads and mutexes in UNIX, `apr_thread_cond_t`'s use pthread conditionals as well.

```
APR_DECLARE(apr_status_t) apr_thread_cond_create(apr_thread_cond_t** cond,
apr_pool_t* pool);
APR_DECLARE(apr_status_t) apr_thread_cond_destroy(apr_thread_cond_t* cond);
APR_DECLARE(apr_status_t) apr_thread_cond_wait(apr_thread_cond_t* cond,
apr_thread_mutex_t* mutex);
APR_DECLARE(apr_status_t) apr_thread_cond_signal(apr_thread_cond_t* cond);
```

11 Appendix B: Differences

This section shows the differences between unmodified Apache HTTP Server 2.2.4 and Pipelined Apache.

```
Index: server/mpm/worker/worker.c
===================================================================
--- server/mpm/worker/worker.c    (revision 7)
+++ server/mpm/worker/worker.c    (revision 30)
@@ -70,6 +70,8 @@
 #include <signal.h>
 #include <limits.h>             /* for INT_MAX */

+#include <pipeline.h>
+
 /* Limit on the total --- clients will be locked out if more servers than
  * this are needed.  It is intended solely to keep the server from crashing
  * when things get out of hand.
@@ -110,6 +112,9 @@
 #define MAX_THREAD_LIMIT 20000
 #endif

+#ifndef MAX_NUM_HANDLERS
+#define MAX_NUM_HANDLERS 20000
+#endif
 /*
  * Actual definitions of config globals
  */
@@ -123,6 +128,7 @@
 static int first_server_limit = 0;
 static int thread_limit = DEFAULT_THREAD_LIMIT;
 static int first_thread_limit = 0;
+static int num_handlers = 1;
 static int changed_limit_at_restart;
 static int dying = 0;
 static int workers_may_exit = 0;
@@ -1135,6 +1141,7 @@
     thread_starter *ts;
     apr_threadattr_t *thread_attr;
     apr_thread_t *start_thread_id;
+    pipeline_t *pipeline;

     mpm_state = AP_MPMQ_STARTING; /* for benefit of any hooks that run as
this
                                    * child initializes
@@ -1181,6 +1188,8 @@
         requests_this_child = INT_MAX;
     }

+    pipeline_create(&pipeline, ap_server_conf, pchild, num_handlers);
+
     /* Setup worker threads */

     /* clear the storage; we may not create all our threads immediately,
@@ -2228,6 +2237,31 @@
     return NULL;
```

```
 }

+static const char *set_num_handlers(cmd_parms *cmd, void *dummy, const char
*arg)
+{
+    const char *err = ap_check_cmd_context(cmd, GLOBAL_ONLY);
+    if (err != NULL) {
+        return err;
+    }
+
+    num_handlers = atoi(arg);
+
+    if (num_handlers > MAX_NUM_HANDLERS) {
+       ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
+                    "WARNING: NumHandlers of %d exceeds compile handler
limit "
+                    "of %d handlers,", num_handlers, MAX_NUM_HANDLERS);
+       ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
+                    " lowering NumHandlers to %d.", MAX_NUM_HANDLERS);
+       num_handlers = MAX_NUM_HANDLERS;
+    }
+    else if (num_handlers < 1) {
+        ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
+                     "WARNING: Require NumHandlers > 0, setting to 1");
+        num_handlers = 1;
+    }
+    ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
+            "num_handlers=%d", num_handlers);
+}
 static const command_rec worker_cmds[] = {
 UNIX_DAEMON_COMMANDS,
 LISTEN_COMMANDS,
@@ -2245,6 +2279,8 @@
   "Maximum number of child processes for this run of Apache"),
 AP_INIT_TAKE1("ThreadLimit", set_thread_limit, NULL, RSRC_CONF,
   "Maximum number of worker threads per child process for this run of
Apache - Upper limit for ThreadsPerChild"),
+AP_INIT_TAKE1("NumHandlers", set_num_handlers, NULL, RSRC_CONF,
+  "Number of handler threads spawned"),
 AP_GRACEFUL_SHUTDOWN_TIMEOUT_COMMAND,
 { NULL }
 };
```

```
Index: server/pipeline.c
===================================================================
--- server/pipeline.c      (revision 0)
+++ server/pipeline.c      (revision 30)
@@ -0,0 +1,742 @@
+#include "pipeline.h"
+#include "apr.h"
+#include "http_log.h"
+
+#define USE_APR_THREAD 1
+
+#if USE_APR_THREAD
+#include "apr_thread_proc.h"
+#else
+#include <pthread.h>
+#endif
+
+extern int do_request_parsing_phase(request_rec *r);
+
+void* APR_THREAD_FUNC request_parsing_phase(apr_thread_t *thd, void *data)
```

```
+{
+    char *curr_phase = "request_parse";
+    char *next_phase = "security";
+    pipeline_t *pipeline= (pipeline_t*)data;
+    request_wrapper *w;
+    int schedule_die = 0;
+
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+    //        "pid %d: Start %s phase", getpid(), curr_phase);
+
+    while (!schedule_die || pipeline->nRequests != 0)
+    {
+        w = NULL;
+        if (!schedule_die)
+        {
+            apr_queue_pop(pipeline->request_parsing_queue, (void*)&w);
+        }
+        else if (apr_queue_trypop(pipeline->request_parsing_queue,
(void*)&w) != APR_SUCCESS)
+        {
+            continue;
+        }
+
+        if (w->die_now)
+        {
+            //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+            //        "pid %d: %s: schedule_die=1",
+            //        getpid(), curr_phase);
+            schedule_die = 1;
+        }
+        else if ((w->access_status = do_request_parsing_phase(w->r))) {
+            ap_log_error(APLOG_MARK, APLOG_ERR, w->access_status, w->r-
>server,
+                    "pid %d, %s: 0x%x error", getpid(), curr_phase, w->r);
+            unlock_wrapper(curr_phase, w);
+        }
+        else {
+            //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+            //        "pid %d: %s: push 0x%x to %s",
+            //        getpid(), curr_phase, w->r, next_phase);
+            //apr_queue_push(pipeline->security_queue, w);
+            apr_queue_push(pipeline->preparation_queue, w);
+        }
+    }
+
+    //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+    //        "pid %d: End %s phase", getpid(), curr_phase);
+    return NULL;
+}
+/*void* APR_THREAD_FUNC security_phase(apr_thread_t *thd, void *data)
+{
+    char *curr_phase = "security";
+    char *next_phase = "preparation";
+    pipeline_t *pipeline = (pipeline_t*)data;
+    int schedule_die = 0;
+
+    request_wrapper *w;
+
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+    //        "pid %d: Start %s phase", getpid(), curr_phase);
+
+    while (!schedule_die || pipeline->nRequests != 0)
+    {
```

```
+        w = NULL;
+        if (!schedule_die)
+        {
+            apr_queue_pop(pipeline->security_queue, (void*)&w);
+        }
+        else if (apr_queue_trypop(pipeline->security_queue, (void*)&w) !=
APR_SUCCESS)
+        {
+            continue;
+        }
+
+        if (w->die_now)
+        {
+            //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+            //        "pid %d: %s: schedule_die=1",
+            //        getpid(), curr_phase);
+            schedule_die = 1;
+        }
+        else if ((w->access_status = do_security_phase(w->r))) {
+            ap_log_error(APLOG_MARK, APLOG_ERR, w->access_status, w->r-
>server,
+                    "pid %d, %s: 0x%x error",
+                    getpid(), curr_phase, w->r);
+            unlock_wrapper(curr_phase, w);
+        }
+        else {
+            //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+            //        "pid %d, %s: push r=0x%x to %s",
+            //        getpid(), curr_phase, w->r, next_phase);
+            apr_queue_push(pipeline->preparation_queue, w);
+        }
+    }
+    //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+    //        "pid %d: End %s phase", getpid(), curr_phase);
+    return NULL;
+}
+*/
+
+#if USE_APR_THREAD
+extern void * APR_THREAD_FUNC do_preparation(apr_thread_t *thread, void
*data);
+#else
+extern int do_preparation(void *data);
+#endif
+
+void* APR_THREAD_FUNC do_preparation_thread(apr_thread_t *thd, void *data)
+{
+    char *curr_phase = "do_preparation_thread";
+    char *next_phase = "";
+    pipeline_t *pipeline = (pipeline_t*)data;
+    int schedule_die = 0;
+
+    request_wrapper *w;
+
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+    //        "pid %d: Start %s phase", getpid(), curr_phase);
+
+    while (!schedule_die || pipeline->nRequests != 0)
+    {
+        w = NULL;
+        if (!schedule_die)
+        {
+            //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
```

```
+            //        "p=%d,%s: waiting for w", getpid(), curr_phase);
+            apr_queue_pop(pipeline->do_preparation_queue, (void*)&w);
+            //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+            //        "p=%d,%s: got w=0x%x", getpid(), curr_phase, w);
+
+            if (!w)
+                continue;
+        }
+        else if (apr_queue_trypop(pipeline->do_preparation_queue,
(void*)&w) != APR_SUCCESS)
+        {
+            continue;
+        }
+
+        if (w->die_now)
+        {
+            //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+            //        "pid %d: %s: schedule_die=1",
+            //        getpid(), curr_phase);
+            schedule_die = 1;
+        }
+        else
+        {
+            do_preparation(NULL,w); /* sets w->access_status */
+            if (w->access_status) {
+                ap_log_error(APLOG_MARK, APLOG_ERR, w->access_status, w->r-
>server,
+                        "pid %d, %s: w=%p req error",
+                        (int) getpid(), curr_phase, w);
+                unlock_wrapper(curr_phase, w);
+            }
+        }
+    }
+    //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+    //        "pid %d: End %s phase", getpid(), curr_phase);
+    return NULL;
+}
+
+void* APR_THREAD_FUNC preparation_phase(apr_thread_t *thd, void *data)
+{
+    char *curr_phase = "preparation";
+    char *next_phase = "handler";
+    int schedule_die = 0;
+
+    int iThread = 0;
+
+    pipeline_t *pipeline = (pipeline_t*)data;
+
+    unsigned int bufsize = 64;
+    char errorBuf[bufsize];
+    apr_status_t rv;
+    int threads_created = 0;
+    unsigned int thread_num = 0;
+
+    const int MAX_THREADS = 10;
+
+    apr_pool_t *thread_pool;
+
+    request_rec r;
+    request_wrapper *w;
+
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+    //        "pid %d: Start %s phase", getpid(), curr_phase);
```

```
+
+#if USE_APR_THREAD
+    apr_thread_t *thread_do_preparation[MAX_THREADS];
+
+    apr_threadattr_t *attr;
+
+    if (apr_pool_create(&thread_pool/*[thread_num]*/, NULL) != APR_SUCCESS)
{
+        ap_log_error(APLOG_MARK, APLOG_ERR, w->access_status, w->r->server,
+                "pid %d, %s: 0x%x: could not start thread pool",
+                getpid(), curr_phase, w->r);
+    }
+
+    apr_threadattr_create(&attr,thread_pool);
+    apr_threadattr_detach_set(attr, APR_DETACH);
+#else
+    pthread_t thread_do_preparation[MAX_THREADS];
+
+    pthread_attr_t attr;
+    pthread_attr_init(&attr);
+    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
+#endif
+
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+    //        "pid %d, %s: Created thread attr", getpid(), curr_phase);
+
+    /* create main request thread pool */
+    for (iThread = 0; iThread < MAX_THREADS; iThread++)
+    {
+        //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+        //        "pid %d, %s: new thread",
+        //        getpid(), curr_phase, next_phase);
+
+        ++thread_num;
+
+#if USE_APR_THREAD
+        if ((rv = apr_thread_create(&thread_do_preparation[iThread], NULL,
+                        do_preparation_thread, (void *) pipeline,
thread_pool)) != 0)
+#else
+        if ((rv = pthread_create(&thread_do_preparation[iThread], NULL,
+                        do_preparation_thread, (void *) pipeline)) != 0)
+#endif
+        {
+#if USE_APR_THREAD
+            char* errorMsg = apr_strerror(rv, errorBuf, bufsize);
+#else
+            char* errorMsg = strerror_r(rv, errorBuf, bufsize);
+#endif
+            ap_log_error(APLOG_MARK, APLOG_ERR, 0, pipeline->server,
+                    "pid %d: %s: can't start thread %d Code %d: %s",
+                    getpid(), curr_phase, threads_created+1, rv,
+                    errorMsg);
+            --thread_num;
+        }
+        //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+        //        "pid %d: %s: threads created=%d",
+        //        getpid(), curr_phase, ++threads_created);
+    }
+
+    while (!schedule_die || pipeline->nRequests != 0)
+    {
+        w = NULL;
```

```
+        if (!schedule_die)
+        {
+            //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server//,
+            //        "p=%d,%s: waiting for w", getpid(), curr_phase);
+            apr_queue_pop(pipeline->preparation_queue, (void*)&w);
+            //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+            //        "p=%d,%s: got w=0x%x", getpid(), curr_phase, w);
+        }
+        else if (apr_queue_trypop(pipeline->preparation_queue, (void*)&w)
!= APR_SUCCESS)
+        {
+            //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+            //        "pid %d, %s: no wrapper, pl->nReqs=%d",
+            //        getpid(), curr_phase, pipeline->nRequests);
+            continue;
+        }
+
+        //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+        //        "pid %d, %s: got w=0x%x, r=0x%x, die_now=%d,
is_closer=%d",
+        //        getpid(), curr_phase, w, w->r, w->die_now, w->is_closer);
+
+        if (w->die_now)
+        {
+            //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+            //        "pid %d: %s: schedule_die=1, pl->nReqs=%d",
+            //        getpid(), curr_phase, pipeline->nRequests);
+            schedule_die = 1;
+        }
+        else if (w->is_closer)
+        {
+            --thread_num;
+            //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+            //        "pid %d, %s: got closer, set thread_num=%d",
getpid(), curr_phase, thread_num);
+            request_wrapper_destroy(w);
+        }
+        else
+        {
+            // subrequest
+            if (w->r->main)
+            {
+                //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+                //        "pid %d: %s: 0x%x do_preparation(subrequest)",
+                //        getpid(), curr_phase, w->r);
+#if USE_APR_THREAD
+                do_preparation(NULL, w);
+#else
+                do_preparation(w);
+#endif
+                unlock_wrapper("preparation", w);
+            }
+            else if (w->r->next)
+            {
+                //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+                //        "pid %d, %s: 0x%x do_preparation(next)",
+                //        getpid(), curr_phase, w->r);
+#if USE_APR_THREAD
+                do_preparation(NULL, w);
+#else
+                do_preparation(w);
+#endif
+                unlock_wrapper("preparation", w);
```

```
+            }
+            else if (w->r->prev)
+            {
+                //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+                //        "pid %d: %s: 0x%x do_preparation(prev)",
+                //        getpid(), curr_phase, w->r);
+#if USE_APR_THREAD
+                do_preparation(NULL, w);
+#else
+                do_preparation(w);
+#endif
+                unlock_wrapper("preparation", w);
+            }
+            // main request
+            else
+            {
+                //ap_log_error(APLOG_MARK, APLOG_DEBUG, w->access_status,
w->r->server,
+                //        "pid %d: %s: r0x%x (main) thread_num=%d",
+                //        getpid(), curr_phase, w->r, thread_num);
+
+                apr_queue_push(pipeline->do_preparation_queue, w);
+                // one already in progress, put back onto queue
+                /*
+                if (thread_num == 1)
+                {
+                    ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline-
>server,
+                            "pid %d, %s: r=0x%x (main), push back into
queue",
+                            getpid(), curr_phase, w->r, next_phase);
+                    apr_queue_push(pipeline->preparation_queue, w);
+                }
+                else if (thread_num == 0) */
+                // something's wrong
+                /*
+                else
+                {
+                    ap_log_rerror(APLOG_MARK, APLOG_ERR, w->access_status,
w->r,
+                            "pid %d, %s: 0x%x: invalid thread_num = %d",
+                            getpid(), curr_phase, w->r, thread_num);
+                }
+                */
+            }
+        }
+    }
+
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+    //        "pid %d: End %s phase, threads_died=%d",
+    //        getpid(), curr_phase, pipeline->threads_died);
+
+    return NULL;
+}
+
+void* APR_THREAD_FUNC handler_phase(apr_thread_t *thd, void *data)
+{
+    thd_arg_t *arg = (thd_arg_t*)data;
+    pipeline_t *pipeline = (pipeline_t*)arg->pipeline;
+
+    int iHandler = arg->iHandler;
+    char *curr_phase = arg->name;
+    char *next_phase = "";
```

```
+    int schedule_die = 0;
+
+    unsigned int bufsize = 64;
+    char errorBuf[bufsize];
+    apr_status_t rv;
+    unsigned int thread_num = 0;
+
+    apr_pool_t *thread_pool;
+
+    request_rec r;
+    request_wrapper *w;
+
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+    //        "pid %d: Start %s phase", getpid(), curr_phase);
+
+    while (!schedule_die || pipeline->nRequests != 0)
+    {
+        w = NULL;
+        if (!schedule_die)
+        {
+            //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+            //        "p=%d/t=%ld %s: blocked on apr_queue_pop",
+            //        getpid(), pthread_self(), curr_phase);
+            apr_queue_pop(pipeline->handler_queue, (void*)&w);
+            //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+            //        "p=%d/t=%ld %s: after apr_queue_pop, w=0x%x",
+            //        getpid(), pthread_self(), curr_phase, w);
+            if (w == NULL)
+            {
+                ap_log_error(APLOG_MARK, APLOG_WARN, 0, pipeline->server,
+                        "pid %d, %s: no wrapper, pl->nReqs=%d",
+                        getpid(), curr_phase, pipeline->nRequests);
+                continue;
+            }
+        }
+        else if (apr_queue_trypop(pipeline->handler_queue, (void*)&w) !=
APR_SUCCESS)
+        {
+            //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+            //        "pid %d, %s: no wrapper, pl->nReqs=%d",
+            //        getpid(), curr_phase, pipeline->nRequests);
+            continue;
+        }
+
+        //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+        //        "pid %d, %s %d: got w=0x%x, r=0x%x, die_now=%d,
is_closer=%d",
+        //        getpid(), curr_phase, iHandler, w, w->r, w->die_now, w-
>is_closer);
+
+        if (w->die_now)
+        {
+            //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+            //        "pid %d: %s: schedule_die=1, pl->nReqs=%d",
+            //        getpid(), curr_phase, pipeline->nRequests);
+            schedule_die = 1;
+        }
+        else
+        {
+            //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+            //        "pid %d, %s: w=0x%x, r=0x%x, call ap_invoke_handler",
+            //        getpid(), curr_phase, w, w->r);
+            w->access_status = ap_invoke_handler(w->r);
```

```
+            //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+            //        "pid %d, %s: w=0x%x, r=0x%x, unlocking wrapper",
+            //        getpid(), curr_phase, w, w->r);
+            unlock_wrapper(curr_phase,w);
+        }
+    }
+    return NULL;
+}
+
+void* APR_THREAD_FUNC log_phase(apr_thread_t *thd, void *data)
+{
+    char *curr_phase = "log_phase";
+    char *next_phase = "";
+    pipeline_t *pipeline= (pipeline_t*)data;
+    request_wrapper *w;
+    int schedule_die = 0;
+
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+    //        "pid %d: Start %s phase", getpid(), curr_phase);
+
+    while (!schedule_die || pipeline->nRequests != 0)
+    {
+        w = NULL;
+        if (!schedule_die)
+        {
+            apr_queue_pop(pipeline->log_queue, (void*)&w);
+        }
+        else if (apr_queue_trypop(pipeline->log_queue, (void*)&w) !=
APR_SUCCESS)
+        {
+            continue;
+        }
+
+        if (w->die_now)
+        {
+            //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pipeline->server,
+            //        "pid %d: %s: schedule_die=1",
+            //        getpid(), curr_phase);
+            schedule_die = 1;
+        }
+        else {
+            ap_run_log_transaction(w->r);
+            unlock_wrapper(curr_phase, w);
+        }
+    }
+    pipeline->threads_died = 1;
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, pipeline->server,
+    //        "pid %d: End %s phase, threads_died=%d",
+    //        getpid(), curr_phase, pipeline->threads_died);
+
+    apr_queue_push(pipeline->die_queue, (void*) 1);
+    return NULL;
+}
+#define PIPELINE_CREATE_DEBUG 0
+
+apr_status_t pipeline_create(pipeline_t **pl,
+                             server_rec *server,
+                             apr_pool_t *pool,
+                             int num_handlers)
+{
+    apr_status_t status;
+    apr_threadattr_t *attr;
+    pipeline_t *pipeline;
```

```
+
+    int i;
+
+    static int max_queue_capacity = 200000;
+
+    pipeline = apr_palloc(pool, sizeof(pipeline_t));
+    *pl = pipeline;
+
+    server->pipeline = pipeline;
+    pipeline->server = server;
+
+#if PIPELINE_CREATE_DEBUG
+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+            "pid %d: creating threadattr", getpid());
+#endif
+    status = apr_threadattr_create(&attr,pool);
+
+#if PIPELINE_CREATE_DEBUG
+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+            "pid %d: setting detach", getpid());
+#endif
+    apr_threadattr_detach_set(attr, APR_DETACH);
+
+#if PIPELINE_CREATE_DEBUG
+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+            "pid %d: creating request_parsing_queue", getpid());
+#endif
+    apr_queue_create(&pipeline->request_parsing_queue, max_queue_capacity,
pool);
+
+    apr_thread_create(&pipeline->request_parsing_thread, attr,
request_parsing_phase, (void*) pipeline, pool);
+#if PIPELINE_CREATE_DEBUG
+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+            "pid %d: created request_parsing_thread=%ld",
+            getpid(), pipeline->request_parsing_thread);
+#endif
+
+    //apr_queue_create(&pipeline->security_queue, max_queue_capacity,
pool);
+//#if PIPELINE_CREATE_DEBUG
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server, "pid %d: creating
security_thread", getpid());
+//#endif
+    //apr_thread_create(&pipeline->security_thread, attr, security_phase,
(void*) pipeline, pool);
+//#if PIPELINE_CREATE_DEBUG
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+            //"pid %d: created security_thread=%ld",
+            //getpid(), pipeline->security_thread);
+//#endif
+
+#if PIPELINE_CREATE_DEBUG
+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+            "pid %d: creating preparation_queue", getpid());
+#endif
+    apr_queue_create(&pipeline->preparation_queue, max_queue_capacity,
pool);
+    apr_queue_create(&pipeline->do_preparation_queue, max_queue_capacity,
pool);
+#if PIPELINE_CREATE_DEBUG
+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+            "pid %d: creating preparation_thread", getpid());
```

```
+#endif
+    apr_thread_create(&pipeline->preparation_thread, attr,
preparation_phase, (void*) pipeline, pool);
+#if PIPELINE_CREATE_DEBUG
+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+            "pid %d: created preparation_thread=%ld",
+            getpid(), pipeline->preparation_thread);
+#endif
+
+    /* Create handler queue */
+#if PIPELINE_CREATE_DEBUG
+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+            "pid %d: creating handler_queue", getpid());
+#endif
+    apr_queue_create(&pipeline->handler_queue, max_queue_capacity, pool);
+
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+    //        "pid %d: created handler_queue=%lu", getpid(), pipeline-
>handler_queue);
+
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+    //        "pid %d: creating %d handler threads", getpid(),
num_handlers);
+    /* Create handler threads */
+    pipeline->handler_threads =
+        apr_palloc(pool,num_handlers * sizeof(apr_thread_t*));
+    for (i=0; i < num_handlers; i++)
+    {
+        thd_arg_t *thd_arg = apr_palloc(pool,sizeof(thd_arg_t));
+        thd_arg->pipeline = pipeline;
+        thd_arg->name = "handler";
+        thd_arg->iHandler = i;
+
+#if PIPELINE_CREATE_DEBUG
+        ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+                "pid %d: creating handler_thread %d", getpid(), i);
+#endif
+        apr_thread_create(
+                &pipeline->handler_threads[i], attr, handler_phase, (void*)
thd_arg, pool);
+        //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+        //        "pid %d: created handler_thread %d=%lu",
+        //        getpid(), i, pipeline->handler_threads[i]);
+    }
+    pipeline->num_handlers = num_handlers;
+
+    /* Create log_thread */
+    //apr_queue_create(&pipeline->log_queue, max_queue_capacity, pool);
+    /* Create log_thread */
+    //apr_thread_create(&pipeline->log_thread, attr, log_phase, (void*)
pipeline, pool);
+
+    /* Create die_queue */
+#if PIPELINE_CREATE_DEBUG
+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+            "pid %d: creating die_queue", getpid());
+#endif
+    apr_queue_create(&pipeline->die_queue, 1, pool);
+
+    /* Set die_now and threads_died */
+    pipeline->die_now = 0;
+    pipeline->threads_died = 0;
+
```

```
+#if PIPELINE_TIME
+    pthread_mutex_init(&pipeline->timer_mutex, NULL);
+#endif
+}
+
+apr_status_t pipeline_die(struct pipeline_t *pl)
+{
+    request_wrapper *w;
+    int i;
+
+    //ap_log_error(APLOG_MARK, APLOG_INFO, 0, pl->server,
+    //        "pid %d: pipeline_die", getpid());
+    pl->die_now = 1;
+
+    /* Send die_wrapper */
+    w = (request_wrapper*) malloc(sizeof(request_wrapper));
+    w->die_now = 1;
+    apr_queue_push(pl->request_parsing_queue, w);
+    //apr_queue_push(pl->security_queue, w);
+    apr_queue_push(pl->preparation_queue, w);
+    for (i=0; i<pl->num_handlers; i++)
+    {
+        apr_queue_push(pl->handler_queue, w);
+    }
+}
+
+void unlock_wrapper(char *phase, request_wrapper *w)
+{
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, w->r->connection-
>base_server,
+    //        " p=%d,%s: w=0x%x unlock_wrapper", getpid(), phase, w);
+
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, w->r->connection-
>base_server,
+    //        "  pid %d,%s: 0x%x Acquiring mutex", getpid(), phase, w->r);
+    apr_thread_mutex_lock(w->mutex);
+
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, w->r->connection-
>base_server,
+    //        "  pid %d,%s: 0x%x Signaling cond", getpid(), phase, w->r);
+    apr_thread_cond_signal(w->cond);
+
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, w->r->connection-
>base_server,
+    //        "  pid %d,%s: 0x%x Unlocking mutex", getpid(), phase, w->r);
+    apr_thread_mutex_unlock(w->mutex);
+    //ap_log_rerror(APLOG_MARK, APLOG_DEBUG, 0, w->r, "  %s: 0x%x Exit
unlock_wrapper", phase, w->r);
+}
+
+#define WRAPPER_CREATE_DEBUG 0
+apr_status_t request_wrapper_create(struct request_wrapper **wrapper,
request_rec *r, int create_mutex)
+{
+    request_wrapper *w = malloc(sizeof(request_wrapper));
+    w->r = r;
+    w->access_status = OK;
+    w->die_now = 0;
+    w->is_closer = 0;
+
+#if WRAPPER_CREATE_DEBUG
+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+            "p=%d: Create wrapper, r=0x%x", getpid(), r);
```

```
+#endif
+
+    if (create_mutex)
+    {
+        /* Create mutex */
+#if WRAPPER_CREATE_DEBUG
+        ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+                "pid %d r=0x%x: Creating mutex", getpid(), r);
+#endif
+        if (apr_thread_mutex_create(&w->mutex, APR_THREAD_MUTEX_UNNESTED,
r->pool) != APR_SUCCESS) {
+            ap_log_error(APLOG_MARK, APLOG_WARNING, 0, r->server,
+                    "pid %d r=0x%x: Could not create mutex", getpid(), r);
+            return APR_EGENERAL;
+        }
+#if WRAPPER_CREATE_DEBUG
+        ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+                "pid %d r=0x%x: Done creating mutex=0x%x", getpid(), r, w-
>mutex);
+#endif
+        /* Create cond */
+#if WRAPPER_CREATE_DEBUG
+        ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+                "pid %d r=0x%x: Creating cond", getpid(), r);
+#endif
+        if (apr_thread_cond_create(&w->cond, r->pool) != APR_SUCCESS) {
+            ap_log_error(APLOG_MARK, APLOG_WARNING, 0, r->server,
+                    "pid %d r=0x%x: Could not create cond", getpid(), r);
+            return APR_EGENERAL;
+        }
+#if WRAPPER_CREATE_DEBUG
+        ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+                "pid %d r=0x%x: Done creating cond=0x%x", getpid(), r, w-
>cond);
+#endif
+    }
+    else
+    {
+        w->mutex = NULL;
+        w->cond = NULL;
+    }
+
+    *wrapper = w;
+
+    return APR_SUCCESS;
+}
+apr_status_t request_wrapper_create_closer(struct request_wrapper
**wrapper)
+{
+    request_wrapper *w = malloc(sizeof(request_wrapper));
+    w->r = NULL;
+    w->mutex = NULL;
+    w->cond = NULL;
+    w->access_status = OK;
+    w->die_now = 0;
+    w->is_closer = 1;
+
+    *wrapper = w;
+
+    return APR_SUCCESS;
+}
+void request_wrapper_destroy(request_wrapper *w)
+{
```

```
+    //if (w->r)
+    //{
+        //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, w->r->server,
+        //        "p=%d,w=0x%x: Destroy wrapper", getpid(), w);
+    //}
+    if (w->mutex) {
+        apr_thread_mutex_destroy(w->mutex);
+    }
+    if (w->cond) {
+        apr_thread_cond_destroy(w->cond);
+    }
+    free(w);
+}
```

```
Index: server/request.c
===================================================================
--- server/request.c       (revision 7)
+++ server/request.c       (revision 30)
@@ -51,6 +51,10 @@
 #include <stdarg.h>
 #endif

+#include "pipeline.h"
+
+#define USE_APR_THREAD 1
+
 APR_HOOK_STRUCT(
     APR_HOOK_LINK(translate_name)
     APR_HOOK_LINK(map_to_storage)
@@ -94,15 +98,15 @@
     }
 }

-/* This is the master logic for processing requests.  Do NOT duplicate
- * this logic elsewhere, or the security model will be broken by future
- * API changes.  Each phase must be individually optimized to pick up
- * redundant/duplicate calls by subrequests, and redirects.
- */
-AP_DECLARE(int) ap_process_request_internal(request_rec *r)
+
+int do_request_parsing_phase(request_rec *r)
 {
     int file_req = (r->main && r->filename);
     int access_status;
+#if PIPELINE_TIME
+    struct timespec end, start;
+    clock_gettime(CLOCK_REALTIME, &start);
+#endif

     /* Ignore embedded %2F's in path for proxy requests */
     if (!r->proxyreq && r->parsed_uri.path) {
@@ -168,7 +172,19 @@
             return access_status;
         }
     }
+    return OK;
+}

+//int do_security_phase(request_rec *r)
+int do_preparation(apr_thread_t *thread, void *data)
+{
+    int access_status;
+    request_wrapper *w = (request_wrapper *) data;
```

```
+    server_rec *server = w->r->server;
+    char *curr_phase = "preparation";
+    request_rec *r = w->r;
+//    int access_status;
+
     /* Skip authn/authz if the parent or prior request passed the
authn/authz,
      * and that configuration didn't change (this requires optimized
_walk()
      * functions in map_to_storage that use the same merge results given
@@ -234,22 +250,132 @@
             break;
         }
     }
+
+//#if USE_APR_THREAD
+//void * APR_THREAD_FUNC do_preparation(apr_thread_t *thread, void *data)
+//#else
+//void * do_preparation(void *data)
+//#endif
+//{
+//    request_wrapper *w = (request_wrapper *) data;
+//    server_rec *server = w->r->server;
+//    char *curr_phase = "preparation";
+//    int access_status;
+
+ //   ap_log_error(APLOG_MARK, APLOG_INFO, w->access_status, server,
+ //           "pid %d, %s: 0x%x: in do_preparation",
+ //           getpid(), curr_phase, w->r);
+//
+//    w->access_status = OK;
+
     /* XXX Must make certain the ap_run_type_checker short circuits mime
      * in mod-proxy for r->proxyreq && r->parsed_uri.scheme
      *                               && !strcmp(r->parsed_uri.scheme,
"http")
      */
-    if ((access_status = ap_run_type_checker(r)) != 0) {
-        return decl_die(access_status, "find types", r);
+    if ((access_status = ap_run_type_checker(w->r)) != 0) {
+        ap_log_error(APLOG_MARK, APLOG_WARNING, w->access_status, server,
+                "pid %d, %s: 0x%x returned access_status=%d",
+                getpid(), curr_phase, w->r, access_status);
+        w->access_status = decl_die(access_status, "find types", w->r);
+        return w->access_status;
     }

-    if ((access_status = ap_run_fixups(r)) != 0) {
-        return access_status;
+    // main request waits here because call ap_run_fixups will spawn a
thread.
+    if ((access_status = ap_run_fixups(w->r)) != 0) {
+        ap_log_error(APLOG_MARK, APLOG_WARNING, w->access_status, server,
+                "pid %d, %s: 0x%x returned access_status=%d",
+                getpid(), curr_phase, w->r, access_status);
+        w->access_status = access_status;
+        unlock_wrapper("fixups", w);
+        //apr_thread_exit(thread, -1);
+        return w->access_status;
     }

+    // if main request, unlock itself.
+    // if subrequest, skip this part.
```

```
+    if (!w->r->main) {
+        request_wrapper *closer;
+        request_wrapper_create_closer(&closer);
+        apr_queue_push(w->r->server->pipeline->preparation_queue, closer);
+        ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+                "pid %d: %s: pushed closer w=0x%x to preparation",
+                getpid(), curr_phase, closer);
+
+        unlock_wrapper("preparation complete", w);
+    }
+
+    // only main request runs on thread
+    if (thread)
+    {
+        ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, server,
+                "preparation: exiting thread %ld", pthread_self());
+        apr_thread_exit(thread, APR_SUCCESS);
+    }
+
     return OK;
 }

+#define PROCESS_REQUEST_INTERNAL_DEBUG 0
+/* This is the master logic for processing requests.  Do NOT duplicate
+ * this logic elsewhere, or the security model will be broken by future
+ * API changes.  Each phase must be individually optimized to pick up
+ * redundant/duplicate calls by subrequests, and redirects.
+ */
+AP_DECLARE(int) ap_process_request_internal(request_rec *r)
+{
+    int access_status = OK;
+    request_wrapper *w;
+    request_wrapper_create(&w, r, 1);

+#if PROCESS_REQUEST_INTERNAL_DEBUG
+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+            "pid %d, Entering ap_process_request_internal %x (%s): %s",
+            getpid(), r, ((!r->main)? "main request" : "subrequest"), r-
>unparsed_uri);
+#endif
+
+
+    /* Lock mutex */
+#if PROCESS_REQUEST_INTERNAL_DEBUG
+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+            "pid %d: 0x%x: Acquiring mutex", getpid(), r);
+#endif
+    apr_thread_mutex_lock(w->mutex);
+
+    /* Push to queue_parse_uri */
+    ap_log_error(APLOG_MARK, APLOG_INFO, 0, r->server,
+            "pid %d: push r=0x%x to request_parsing_queue, pl->nReqs=%d",
+            getpid(), r, r->server->pipeline->nRequests);
+    apr_queue_push(r->server->pipeline->request_parsing_queue, w);
+
+    /* Wait */
+#if PROCESS_REQUEST_INTERNAL_DEBUG
+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+            "p=%d,w=0x%x: Wait on cond", getpid(), w);
+#endif
+    apr_thread_cond_wait(w->cond, w->mutex);
+    /* Got unlocked */
+#if PROCESS_REQUEST_INTERNAL_DEBUG
```

```
+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+            "p=%d,w=0x%x: Unlock mutex", getpid(), w);
+#endif
+    apr_thread_mutex_unlock(w->mutex);
+
+#if PROCESS_REQUEST_INTERNAL_DEBUG
+    ap_log_error(APLOG_MARK, APLOG_INFO, 0, r->server,
+            "pid %d, Exiting ap_process_request_internal w=0x%x (%s) w/
access_status=%d",
+            getpid(), w, ((!r->main)? "main request" : "subrequest"), w-
>access_status);
+#endif
+    if (w->access_status) {
+        access_status = w->access_status;
+    }
+    request_wrapper_destroy(w);
+    ap_log_error(APLOG_MARK, APLOG_INFO, 0, r->server,
+            "pid %d: end internal r=0x%x, pl->nReqs=%d",
+            getpid(), r, r->server->pipeline->nRequests);
+    return access_status;
+}
+
+
 /* Useful caching structures to repeat _walk/merge sequences as required
  * when a subrequest or redirect reuses substantially the same config.
  *
```

Index: modules/http/http_core.c

```
===================================================================
--- modules/http/http_core.c     (revision 7)
+++ modules/http/http_core.c     (revision 30)
@@ -35,6 +35,8 @@

 #include "mod_core.h"

+#include "http_log.h"
+
 /* Handles for core filters */
 AP_DECLARE_DATA ap_filter_rec_t *ap_http_input_filter_handle;
 AP_DECLARE_DATA ap_filter_rec_t *ap_http_header_filter_handle;
@@ -172,7 +174,8 @@
      * Read and process each request found on our connection
      * until no requests are left or we decide to close.
      */
-
+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, c->base_server,
+    //        "pid %d: ap_process_http_connection Enter", getpid());
     ap_update_child_status(c->sbh, SERVER_BUSY_READ, NULL);
     while ((r = ap_read_request(c)) != NULL) {

@@ -203,6 +206,8 @@
         /* Go straight to select() to wait for the next request */
     }

+    //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, c->base_server,
+    //        "pid %d: ap_process_http_connection Exit", getpid());
     return OK;
 }
```

Index: modules/http/http_request.c

```
===================================================================
--- modules/http/http_request.c  (revision 7)
+++ modules/http/http_request.c  (revision 30)
```

```
@@ -50,6 +50,8 @@
 #include <stdarg.h>
 #endif

+#include "pipeline.h"
+
 /* ********************************************************************
  *
  * Mainline request processing...
@@ -234,7 +236,11 @@
 void ap_process_request(request_rec *r)
 {
     int access_status;
+    char* phase = "ap_process_request";

+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+            "pid %d, %s: 0x%x Enter", getpid(), phase, r);
+
     /* Give quick handlers a shot at serving the request on the fast
      * path, bypassing all of the other Apache hooks.
      *
@@ -253,10 +259,38 @@
         ap_time_process_request(r->connection->sbh, START_PREQUEST);
     access_status = ap_run_quick_handler(r, 0);  /* Not a look-up request
*/
     if (access_status == DECLINED) {
+        ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+                "pid %d, %s: Call ap_process_request_internal(r=0x%x)",
getpid(), phase, r);
+
+        /* Increment pipeline requests */
+        ++r->server->pipeline->nRequests;
+
         access_status = ap_process_request_internal(r);
         if (access_status == OK) {
-            access_status = ap_invoke_handler(r);
+            //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+            //        "pid %d, %s: call request_wrapper_create(r=0x%x)",
+            //        getpid(), phase, r);
+            request_wrapper *w;
+            request_wrapper_create(&w,r, 1);
+
+            //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+            //        "pid %d, %s: w=(0x%x) lock w->mutex",
+            //        getpid(), phase, w);
+            apr_thread_mutex_lock(w->mutex);
+            //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+            //        "pid %d, %s: push(w=0x%x) to handler_queue=%p",
+            //        getpid(), phase, w, r->server->pipeline-
>handler_queue);
+            apr_queue_push(r->server->pipeline->handler_queue, w);
+            apr_thread_cond_wait(w->cond, w->mutex);
+            apr_thread_mutex_unlock(w->mutex);
+            //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+            //        "p=%d/t=%ld %s: (w=0x%x) handler mutex unlocked",
+            //        getpid(), pthread_self(), phase, w);
+
+            request_wrapper_destroy(w);
         }
+        /* Decrement pipeline requests */
+        --r->server->pipeline->nRequests;
     }
```

```
     if (access_status == DONE) {
@@ -284,6 +318,9 @@
     ap_run_log_transaction(r);
     if (ap_extended_status)
         ap_time_process_request(r->connection->sbh, STOP_PREQUEST);
+
+    ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, r->server,
+            "pid %d, %s: 0x%x Exit", getpid(), phase, r);
 }

 static apr_table_t *rename_original_env(apr_pool_t *p, apr_table_t *t)
```

```
Index: include/pipeline.h
===================================================================
--- include/pipeline.h     (revision 0)
+++ include/pipeline.h     (revision 30)
@@ -0,0 +1,80 @@
+#ifndef __PIPELINE_H__
+#define __PIPELINE_H__
+#include "apr_thread_proc.h"
+#include "apr_queue.h"
+
+#include "httpd.h"
+#include "mpm_common.h"
+
+#define TWO_HANDLERS 1
+
+struct pipeline_t
+{
+    apr_thread_t *request_parsing_thread;
+    apr_queue_t *request_parsing_queue;
+
+    //apr_thread_t *security_thread;
+    //apr_queue_t *security_queue;
+
+    apr_thread_t *preparation_thread;
+    apr_queue_t *preparation_queue;
+    apr_queue_t *do_preparation_queue;
+
+    apr_thread_t **handler_threads;
+    apr_queue_t *handler_queue;
+
+    apr_thread_t *log_thread;
+    apr_queue_t *log_queue;
+
+    server_rec *server;
+
+    int die_now;
+
+    /** Used to have child wait for threads when exiting */
+    apr_queue_t *die_queue;
+
+    /** Number of requests in the pipeline used to know when to exit. */
+    int nRequests;
+
+    /** If all threads ended */
+    int threads_died;
+
+    int num_handlers;
+};
+typedef struct pipeline_t pipeline_t;
+
+/** Used to pass info to handler_phase. */
```

```
+struct thd_arg_t {
+    pipeline_t* pipeline;
+    char* name;
+    int iHandler; /* handler index to handler_threads */
+};
+typedef struct thd_arg_t thd_arg_t;
+
+apr_status_t pipeline_create(struct pipeline_t **pl,
+                             server_rec *server,
+                             apr_pool_t *pool,
+                             int num_handlers);
+apr_status_t pipeline_die(struct pipeline_t *pl);
+
+#include "apr_thread_cond.h"
+
+struct request_wrapper {
+    request_rec *r;
+    int access_status;
+    apr_thread_mutex_t *mutex;
+    apr_thread_cond_t *cond;
+    int die_now;
+    int is_closer;
+};
+typedef struct request_wrapper request_wrapper;
+
+apr_status_t request_wrapper_create(struct request_wrapper **w, request_rec
*r, int create_mutex);
+apr_status_t request_wrapper_create_closer(struct request_wrapper
**wrapper);
+
+void request_wrapper_destroy(request_wrapper *w);
+
+void unlock_wrapper(char *phase, request_wrapper *w);
+
+#endif
+
```

```
Index: include/httpd.h
===================================================================
--- include/httpd.h (revision 7)
+++ include/httpd.h (revision 30)
@@ -1218,6 +1218,9 @@

     /** The server request scheme for redirect responses */
     const char *server_scheme;
+
+    /** Pipeline threads */
+    struct pipeline_t *pipeline;
 };

 typedef struct core_output_filter_ctx {
```